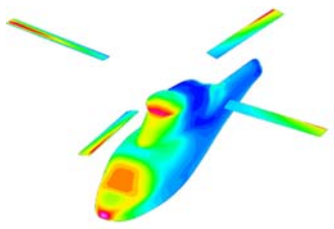
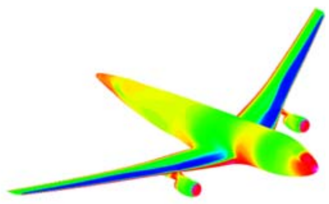
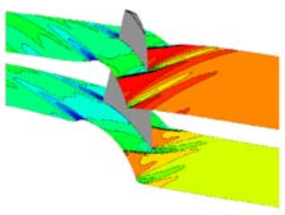
*elsA*xdt User Manual

Python/CGNS interface for *elsA* – Release v5.30







Quality	Author	For the reviewers	Approver
Function		Quality manager	Project head
Name	M. Poinot	A.-M. Vuillot	S. Heib
Visa	*Signé par M. Poinot*	*Signé par A.-M. Vuillot*	*Signé par S. Heib*

Software management : ELSA SCM
Applicability date : immediate
Diffusion : Authors, *elsA* users

HISTORY

version edition	DATE	CAUSE and/or NATURE of EVOLUTION
2.0	January 04, 2010	Edition for release 5.30 of the Python/CGNS interface for *elsA*
1.1	July 19, 2007	Edition for release 5.3 of the Python/CGNS interface for *elsA*
1.0	May 30, 2002	Creation

ONERA

THE FRENCH AEROSPACE LAB

elsAxdt

Guide to CGNS/Python interface for elsA

Marc POINOT
ONERA/CS2A

/ELSA/MU-02052/v2.0
January 2010

The *elsAxdt* module is the Python/CGNS interface for the *elsA* CFD solver kernel. It is a standard Python module that can be imported from usual Python scripts. The *elsAxdt* module is required for CGNS-based computation or code-coupling with *elsA*.

This document details the interface of the *elsAxdt* module, the input CGNS trees the module can translate for *elsA* and the output CGNS trees that can be returned from the *elsA* computations.

1. FOREWORDS

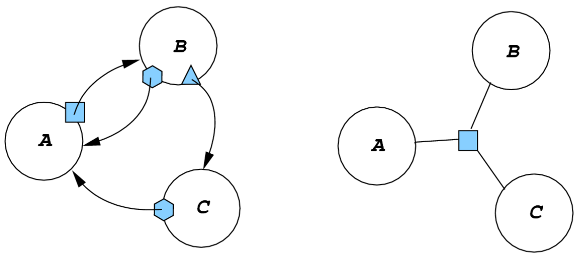
A part of the *elsA* CFD solver interface is available through a Python module named *elsAxdt*. It performs solver input/output using plain Python data structures and uses the CGNS data model as logical description of information. We explain in this document how to use the *elsAxdt* interface, and thus how to use *elsA* with CGNS.

1.1 CGNS Standard

The *elsA* solver is a Navier-Stokes fluid solver, it can be used on his own or integrated into a more complex multi-physics simulations. A way to insure interoperability between the codes involved in such a complex simulation is to use a standard. The CFD community has a standard for data model definition and disk storage: CGNS (CFD General Notation System).

1.1.1 Public standards

The use of a public data standard opens a door to other software customers and providers. You can invent your own standard for a given proprietary simulation, but you cannot use third party tools. It is more efficient for a team, in terms of developpement and maintenance, to learn and add a CGNS adapter for everybody, than to add one translator for each possible code or framework. A public standard also increases the user community and thus increases the experts you can find, the documentation, tutorials, examples, tools...



Many data formats are restricted to plain float arrays or even discrete data. Thus, the user has to be aware of the meaning and the structure of the data before any use of the actual values. The CGNS standard is a data specification (or data model), it comes with a detailled and well defined meaning for every part of the data structure.

Such a standard could be seen as a restriction to proprietary formats, because many of these formats are exactly defined for the targeted applications. The standard is more a consensus, the largest common set of agreements between people making solvers, making tools or using them. Then, while not perfect, the CGNS standard is the ground for *elsA* data model interface.

1.1.2 CGNS version

The reference document for CGNS is the so-called SIDS, which stands for Standard Interface Data Structure. The reference version is v2.3.4 which is described in the document AIAA-R101A-2006 publicly available on http://www.cgns.org (official CGNS web site).

1.2 Storage vs data model

A data model is a logical view of data. Such a view can be mapped to a physical view. For instance, saying that a 3D point with cartesian coordinates have three coordinates X,Y and Z is a logical (or conceptual) view of a point. An array with three 64 bits floating point numbers is a physical view of the same point.

In this document, the data specifications (or models) are represented as trees. That is a node with a set of nodes so-called children, and so on... Each node has a type, a name, a value and a list of children. We detail in this manual all the trees *elsA* can read, write or even update during run-time. These trees are logical views of data and you need an actual physical mapping to use them in your application. For this mapping, the *elsAxdt* provides a Python representation of the tree data (which is the default in this package). In that case the data are kept in memory as 100% Python objects with pointers to actual *elsA* arrays in memory. The *elsAxdt* interface provides means to store this Python/CGNS tree on disk. The physical representation is then the one provided by your own CGNS library. In other words, if you have a CGNS/HDF5 library and you use it in the *elsA* production, the actual physical files will be HDF5 files. If you have a CGNS/ADF library, the *elsA* files will be ADF files.

All these translations are easy to use, these are not time nor memory consuming and insure a good consistency for all simulation co-workers.

1.3 Python arrays implementation

You should take care of the actual Python array implementation you are using. The *elsAxdt* module can be used with one of the three usual arrays modules: Numeric, numarray and numpy. The numpy module is the default and strongly recommended array module. When *elsAxdt* is imported, a message is sent to the standard output. This message recalls you the Python array module which has been used for *elsAxdt* installation. You must insure this array module is the same as the one used by *elsA* or any other Python module you plan to import in your Python scripts.

1.4 About this document

The *elsAxdt* manual is, at the same time, a reference document and a users' guide. As we are supporting the CGNS standard, many informations related to the standard are not detailled here. The user should read the SIDS to have more complete information about the CGNS data structures.

The chapters of this document are:

- ***Quick Start*** to be able to run *elsAxdt* on examples in five minutes.
- ***Interface*** describes the objects and functions available through *elsAxdt*, in order to drive *elsA* and to use CGNS.
- ***Profile*** is the reference for CGNS trees *elsAxdt* can understand or produce.
- ***Package*** is a technical chapter, describing the module architecture and some installation remarks.

The features, examples and interface detailled in this document are based on at least these versions:

elsAxdt	**v5.30**
elsA	**v3.3**

If you have an release with a greater version number, compatibility is insured.

2. QUICK START

You want to use *elsAxdt* now! Follow the small tutorial hereafter, we assume you already have installed the *elsAxdt* Python module. If this is not the case, please refer to the chapter Package where a section explains the installation.

2.1 The module import

We are writing a Python script., it can be run using *Python* or *elsA.x* as interpreters the example listed here after can be run with both but we recommend to use the *elsA.x* interpreter, the required shell environment to run it is easier to set than the plain Python environment.

The first statement you have to write in your Python code is the *elsAxdt* import. The module has few functions you can call. For example, in the case of a parallel computation, you might want to know which are the current process number of your running program and how many processes there are in your computation.

```
import elsAxdt
(rank,size)=elsAxdt.ranksize()
```

You write these two lines in a `myscript.py` file, you run it with the command:

```
elsA.x myscript.py
```

You can do the same with the help of an MPI Python module, *elsAxdt* provides this information to allow you to run a parallel computation without importing other modules, but this is restricted to these two usual values *`rank`* and *`size`*.

△ If you have some error return at this point, you should read the section 6.3 (Known issues and common troubleshootings) in this manual.

△ If you don't use MPI, *`ranksize()`* always returns the value `(0,1)`.

△ You MUST use a specific Python interpretor or *`mpirun`* to run a parallel script.

2.2 Reading a CGNS file

We want now to read a CGNS file for our computation. This file is the input tree. We call that a tree because it is organized with nodes having children, each node is compliant with the CGNS data model and the whole tree itself is compliant. We create an xdt tree in memory using the CGNS file given as argument. The variable *`mytree`* is the xdt representation of the *`001Disk.cgns`* tree. The *`mytree`* object can be used to query some practical informations about the loaded tree.

```
1 import elsAxdt
2 mytree=elsAxdt.XdtCGNS("001Disk.cgns")
3 print mytree.filename
```

The CGNS file can contain links to other CGNS files, it is up to you to insure that all required files are available in the correct directory. During the read, the links will be followed, the whole data is gathered into a single in-memory representation of all the computation data you have put in the CGNS file.

△ We strongly suggest you use only file names without any relative or absolute path reference in your link paths.

△ You can specify a directory search path for the linked-to files, read the section 6.3.

The actual read is not performed in line 2, we see in next section how to perform the actual read and to run the computation.

2.3 Running elsA

The *elsA* solver requires a complete simulation description. You can define all the simulation data and control parameters in a single CGNS file, or define only a part of these information in a CGNS file and complete the description in a Python script with *elsA* legacy commands. This latter mode is so-called *migration* mode. The example below shows how to run a full CGNS computation.

```
1 import elsAxdt
2
3 mytree=elsAxdt.XdtCGNS("001Disk.cgns")
4 mytree.mode=elsAxdt.READ_ALL
5
6 resulttree=mytree.compute()
7 mytree.save("001Disk-output.cgns")
```

The line 4 sets a flag on `mytree` to indicate that all the CGNS tree has to be read. The line 6 actually runs the computation, the elsA solver should find in `mytree` (i.e. in the CGNS tree) all the required data. The result of the computation is returned as a new *xdttree*, containing all requested output (such as result fields, convergence). The layout of a classical CGNS tree for a complete computation is detailed in the next section.

This result is not saved in a file unless you ask elsAxdt to do so. The line 7 saves the result in a CGNS file. This file doesn't contain the initial computation data such as grids, reference state... You have to use CGNS tools to add links from your resulting output CGNS file to the initial CGNS file in order to have grid information (this is required for visualization tools for example).

△ The target object for the `save` method is `mytree`, not `resulttree`. Because `resulttree` is a plain Python object and it has no specialized methods such as `save`. This `resulttree` can be used in your Python script for any purpose your application would like to.

2.4 CGNS trees layout

The main script looks very easy to write. Actually, this script only contains the read and the write of the CGNS tree. As you may note, the output tree is not defined in any way. There is a default output tree. If you do not specify a specific output, the returned tree contains conservative values for all zone fields with the associated convergence per zone. You have to refer to next sections if you want a complete description of input/output CGNS tree for *elsAxdt*.

The input tree is never modified, it is taken as read-only by *elsAxdt*, the output tree is generated from scratch with the same base/zone structure but it does not contain the grid coordinates. As explained in the next sections, you have to make links to the grid coordinates by yourself.

One should understand that the tree in memory once you have performed a 'read' is not a Python tree, but rather an xdt tree. This means you cannot use the `mytree` object with any kind of Python functions, you can only use the xdt class interface. You can have the same tree as a 100% Python objects tree, you use the `hollow` method on your xdt tree instance.

```
import elsAxdt
mytree=elsAxdt.XdtCGNS("001Disk.cgns")
scopetree=mytree.hollow()
print scopetree
```

The variable `scopetree` is a list of lists, representing a Python tree or CGNS/Python tree (see section 5.5.4 SCOPE). This Python tree is a plain Python objects tree, it has no proprietary classes and only requires the *numpy* module for actual arrays. This CGNS tree representation can be understood by any Python script without required import.

The node values, such as `DataArray_t` *values,* are pointers to the same memory zone as your xdt tree shared with the actual *numpy* arrays. This means that, for example, if you read large arrays of point coordinates, these will not be duplicated when you ask for a Python "*hollow*" view of your xdt tree.

3. MODULE USER INTERFACE

The interface is first composed of a Python package with functions, classes and other Python services, and second with the description of the expected CGNS trees as input and produced CGNS trees as output. The CGNS trees structure and the meaning of these structures are not explained in details in this chapter, but in the next chapter.

This chapter describes the Python package interface and some helpful hints for dealing with the Python/CGNS trees. We assume you are familiar with both Python and CGNS tree structure.

3.1 Python package interface

The user interface is a Python module. The whole elsa solver is loaded as a module, the main class interface is the xdt class. Keep clear in your mind that there is on one side the Python module, so-called elsAxdt and on the other side the xdt class. This class is defined by elsAxdt and belongs to the interface, but it actually is something else that the package itself.

3.1.1 The *elsAXdt* module

The module can be imported from a plain Python interpreter or from the elsA solver itself. Make sure your interpreter or your solver has the parallel capabilities if you want to use the MPI services. The module level functions and constants can be used to get/set information before the actual creation of an xdt object

- ☐ *`args(list-of-strings)`* is used to set the command line argument and pass it to the solver. In the case you are using elsAxdt from a plain Python interpreter, you have to mimic the command line argument for the elsA solver. This function is mandatory in the case of code-coupling.
- ☐ *`iteration()`* is a function call to the elsA solver which returns the current iteration number. This is used in the scope of code-coupling applications, when your coupling script wants to know at which iteration it is performed.
- ☐ *`get(tree-name-as-string)`* returns a solver CGNS tree with the given name, if found. The returned tree is a Python/CGNS tree. The reserved names are `INPUT_TREE` (the tree you read at first), `OUTPUT_TREE` (the tree used to store all output results) and `RUNTIME_TREE` (the tree used as I/O shuttle during code-coupling computations).
- ☐ *`output()`* returns the current output CGNS tree found in the solver. This can be performed without creation of an xdt object. The returned tree is a Python/CGNS tree. Same as `get(OUTPUT_TREE)`.
- ☐ *`list()`* returns the list of the names of the CGNS trees currently in the solver.
- ☐ *`ccname()`* returns the name of the elsA instance (i.e. the '`-c`' option value).
- ☐ *`state()`* returns the current state of the solver (see section 5.2.2 State stops).
- ☐ *`ranksize()`* is used for parallel application, it returns a tuple of integers (rank, size) required for parallel distribution of tasks amongst the processors. The call can be performed even in sequential mode (i.e. without a call to the MPI library), in that case the tuple (0,1) is returned.

The use of the `arg` function usually is made with the `sys.argv` variable. In the context of code-coupling, the pattern shown in the code snippet below is mandatory because elsA is expected command line args for the code-coupling parameters:

```
import elsAxdt
import sys

elsAxdt.args(sys.argv)
```

There is another function, xdt, used for a new xdt object creation. You can create such an object using this xdt call with the required arguments, or you may prefer to use the interfaces provided by the Xdt classes. The next section describes these classes.

3.2 *The Xdt class*

An Xdt tree is C++ tree with a CGNS tree layout (i.e. the organization of the nodes, their names, etc...). The elsA~ solver can run a computation using all or a part of the data it can found in this tree. The overall structure of the tree is explained in the next chapter, we assume here you have a correct tree available.

You have several ways to create and modify an Xdt tree, depending on your application, you may have a complete tree in a CGNS file or you may use full Python trees for in-memory code-coupling.

An Xdt object is created with different arguments depending on the adapter. An adapter is a translator which helps you to create or copy trees. Five adapters are available, but two of them are really useful with the current solver version:

- ☐ *PYTHON* uses the Python CGNS tree representation
- ☐ *CGNS* uses the CGNS/HDF or ADF tree on disk

Depending on the adapter, the arguments are different. The most simple adapter is the CGNS one which reads a CGNS file.

```
mytree=XdtCGNS("010Disk.cgns")
```

The file is not read, the *mytree* object is created but it waits for more options before reading the file.

```
mytree=XdtCGNS("010Disk.cgns")
mytree.load(READ_ALL)
```

Now the file is read, the complete file as specified by the *READ_ALL* option. You can perform the same command in a single line:

```
mytree=XdtCGNS("010Disk.cgns").load(READ_ALL)
```

At this time, you have in memory a C++ Xdt tree with the structure and the data of your CGNS file. You can also ask to a Xdt tree to save itself into a CGNS file:

```
mytree.duplicate("010Disk.cgns")
```

The Xdt tree knows it has to save as a CGNS file because it was created wich a CGNS adapter, in other words mytree is an Xdt tree with CGNS input/ouput capabilities.

> △ Please do not mismatch save and duplicate functions. For backward compatibility concerns, the save function is reserved for the computation result (i.e. the output tree), while duplicate actually saves the current tree.

In some complex computation cases, for exemple in Chimera, you may have to parse different trees, each having a part of the simulation grids. The add function purpose is to read a CGNS tree and to add it to the existing CGNS trees. In the next example, `gfiles` is a list of files, the first file is used to create the Xdt object, then the other files are parsed, all the `CGNSBase_t` node are taken into account.

```
mytree=xdt.XdtCGNS(gfiles[0])
for gfile in gfiles[1:]:
 mytree.add(gfile)
```

Another useful adapter is the Python adapter. The idea is the same as the previous CGNS adapter, in that case the Xdt tree knows how to do input/output with Python objects. Then you have to represent a CGNS tree using Python (see 5.5.4):

```
scopetree=[['CGNSLibraryVersion', 2.4, [], 'CGNSLibraryVersion_t'],
           ['SquaredNozzle-09', array([3, 3]), [], 'CGNSBase_t']]
mytree=XdtPython(scopetree)
```

The CGNS tree we use in this example only as a *CGNSBase_t* node, and it is quite tedious to define a complete tree by hand. Thus you can use the pyCGNS Python module to create and process Python/CGNS trees. This module is not required by elsAxdt, but it would help if you want to do some operations on the Python/CGNS trees.

The output of a Python adpater is a Python object, it stays in the Python interpreter memory and you can process it with usual Python methods. The arrays in the tree are numpy arrays. If you want to know more about large array memory management please read the section .

△ The numpy Python arrays require you to define the array with explicit integer types when you use the ASCII representation of an array: `array([3,3],'i')`

You may also want to go from one representation to another one. For example you have a CGNS adapter but you want a Python representation. You use the hollow method:

```
mytree=XdtCGNS("010Disk.cgns")
pytree=mytree.hollow()
```

3.3 Adapters methods

The adapter base class is XdtParser, the other adapters such as XdtCGNS and XdtPython are derived class. You can always use the base class methods, but the specialized classes have syntactic sugar that can simplify your code readability and reduce typo errors.

3.3.1 XdtParser

The base class of the adapters, it provides methods for creating a tree, setting options, getting parameters or saving the tree. You also have special functions that actually run the elsA solver or interact with it. The enumerates or constants defined in the module are detailled in a next section. For example, the flags you can give as arguments to the load method should be of enumerate LoadMode. Other parameters are usual Python types, such as string, integers, etc...

- ☐ *`XdtParser`* creates a new object, arguments are the adapter type and the parameters. The parameters type change depending on the adapter type, for exemple a CGNS adapter will require the filename where the Python adapter waits for a Python CGNS tree (i.e. a list of nodes).
- ☐ *`load(flags)`* actually creates the Xdt tree taking into account the set of flags passed as arguments (LoadMode with default value `READ_ALL`). The return value is the Xdt object itself.
- ☐ *`compute()`* starts the elsA computation. The return value is a new Xdt object with the resulting computation tree.
- ☐ *`save(filename)`* saves the output Xdt object in a CGNS file. All the tree is stored in a single file. If the filename already exists, the adapter automatically adds a trailing `#` to the filename and tries again to save until it succeed. Thus, if you run several times your script, you will have many `#` files and the newest one has the highest number of `#`.
- ☐ *`hollow()`* returns a Python/CGNS tree representation of the current Xdt object.
- ☐ *`dump(a,p1,p2)`* creates a new Xdt object with adapter a (Adapter) and parameters p1 and p2. CGNS file with name filename (string).
- ☐ `add(filename)` reads a CGNSTree and adds the CGNSBase to the already parsed file(s).

In addition with these methods, the adapters have specific methods for the migration purpose. See next section for these specific methods.

The class has the following attributes (some of them are automatically updated but you can overwrite the values).

☐ *`mode`*	flags to drive the parts of the tree to read.
☐ *`selection`*	list of Zones to read (list).
☐ *`distribution`*	association Zone/processor number (dictionary).
☐ *`debug`*	level of debug trace

3.3.2 XdtPython

The Python adapter is a specialized XdtParser that can use a CGNS Python tree as entry. This adapter is used when you want to create or modify CGNS trees during run-time, or depending on specific parameters you may only know at run-time. This is the required adapter for code-coupling with the solver, since we do not use the file storage for data exchange but rather the actual data in memory.

XdtPython creates a new object, argument is the Python object (i.e. a list) to use as tree.

The pyCGNS module uses the Python in-memory CGNS tree. You can load/store trees on CGNS files and use many useful methods for your Python scripts. This pyCGNS module is not included in elsAxdt, it is a stand alone Open Source Python module.

> △ This class interface is not stable and we do not recommand to use this class in your operational scripts.

3.3.3 XdtCGNS

This adapter is used when you use a CGNS file. You should give the complete file name with the extension, the adapter does not care if you have an `.adf`, `.hdf` or `.cgns` file. However, the file format you use should be the one the library you used during elsAxdt production.

An *XdtCGNS* adapter has a specialized attribute filename which contains the name of the CGNS file used.

- ☐ *`XdtCGNS`* creates a new object, argument is the filename.

3.4 Migration methods

The following methods can be applied to an Xdt instance. The methods with the migration tag are only useful if you want to mix a legacy elsa Python script and an Xdt tree parse.

- ☐ *`elsA2CGNS(elsaname)`* returns the CGNS name (string) as a path to the actual CGNS node used to create the elsA object with the name given as argument (string).
- ☐ *`CGNS2elsA(cgnsname)`* returns the elsA name (string) in the case this CGNS node given as argument (string) was used to create an elsA object. Otherwise, raises an error.
- ☐ *`symboltable()`* returns all the elsA names created from a CGNS node.
- ☐ *`e_forceValue(name,attribute,value)`* set the value (integer,float,string) to the internal attribute (string) of the object with internal name (string). As the operation is performed on the actual elsA objects, the method should be used with care. No control of any kind is made on the attribute name nor the value.
- ☐ *`e_getValue(name,attribute)`* get the value (integer,float,string) from the internal attribute (string) of the object with internal name (string). The attribute name should be defined on the target object, unless the operation fails during run-time, no control is made on the attribute name.

These migration methods have pure CGNS names or pure elsa names as arguments, you should refer to the documentation before any attempt to retrieve an actual object or its name from the elsa solver. We suggest you use the migration patterns you can find in this document or in the module examples.

Some more migration methods exist, however we **do not recommend** to use them. The use of migration methods can be dangerous is you set bad parameters or if you want to access to an object before it is actually created. If you really need a migration method and you cannot use one in the list above, please contact elsA support.

3.5 Module constants

Some predefined constants are set in the module dictionnaries. These constants are enumerates and can be found in two dictionaries. For example, you use the constant CGNS as a function argument:

```
a=elsAxdt.CGNS
```

There are cases you only have the enumerate value itself (i.e. 4) but you want to know which constant it is, you use the look back dictionary:

```
a=elsAxdt.CGNS
b=elsAxdt.Adapter['CGNS']
c=elsAxdt.Adapter_[elsAxdt.CGNS]
```

A reverse lookup dictionnary has the dictionnary name with a trailing underscore. The module dictionnaries are described in the section hereafter. All dictionnaries of the elsAxdt module are in the `elsAxdt.Enumerates` top dictionnary.

We suggest you to use these constants, then your code would have less dependency to a change to the module implementation. We detail here these constants.

3.6 Enumerates and constants Dictionaries

The values listed here are defined as Python. We encourage you to use these constants instead of actual string or integer values. Moreover, we strongly suggest you always prefix the constants with the module name,

3.6.1 Adapter

The Adapter enumerates are parameters to the Xdt class translator. That is, when you create an Xdt object using the base class, you have to specify what kind of adapter you want.

- ☑ *PYTHON* Xdt/Python translator, read/writes Python objects in memory.
- ☐ *CGNS* Xdt/CGNS translator, read/writes CGNS/ADF or CGNS/HDF5 files.
- ☐ *USER, XML, MEMORY* Reserved for future usage

Actually, the CGNS and PYTHON adapters are the most used, and you would prefer to create Xdt objects using the specialized classes and not the base one.

3.6.2 ErrorCode

The error enumerate gives you the level of the returned error.

- ☑ *OK* Nothing bad happens.
- ☐ *WARNING* Something bad happens, but this is not that important, we just inform you on this and you continue to work.
- ☐ *ERROR* Something wrong happens, but you can decide to continue to work.
- ☐ *FATAL* Something wrong happens, no way for elsAxdt to continue to work, you have to stop.

3.6.3 ParseMode

The parse mode tells the Xdt object what kind of operation to perform after reading the CGNS tree.

- ☑ *COMPUTE* Once the tree is read, run the actual solver computation.
- ☐ *TRANSLATE* Load the tree but do not run the computation.
- ☐ *UPDATE* Update the tree instead of loading it.

These three enumerates can be set to the action attribute or you can use the methods compute, translate and update instead.

3.6.4 LoadMode flags

These flags indicates which parts of the CGNS tree have to be taken into account. This is mainly used for elsA ~legacy scripts migration or tuning. Each flag can be or-ed to build a bit string with options. The default value for this parameter is READ_ALL.

☐ *READ_MESH*	Read Grids (no connectivity)
☐ *READ_CONNECT*	Read connectivities
☐ *READ_BC*	Read boundary conditions

☐ *READ_INIT*	Read initialisation fields
☐ *READ_FLOW*	Read flow equation sets (not flow solutions !)
☐ *READ_COMPUTATION*	Read computation sub-tree
☐ *READ_OUTPUT*	Read output skeletton
☐ *READ_TRACE*	Set the trace mode
☐ *READ_NONE*	Empty option set
☐ *READ_ALL*	Full options set

The elsAxdt module loads a CGNS tree as a trustable and consistent set of data. Thus there are not a lot of verifications and if an error occurs during the tree parse, it often is a fatal error. You can drive the tree parsing by saying wihch parts of the global tree you want to take into account or not. This is the LoadMode set of flags. Each flag can be added to the actual flags you pass to the tree parser:

```
myflag=READ_MESH
myflag=myflag | READ_CONNECT

myflag |= READ_BC
```

Here you say you want to read the meshes and the connectivities of the argument tree. All other parts of the tree will be ignored. The two syntaxes for setting the flag are the same. Please refer to the Python manual, the flags are true Python objects. If you want to know exactly what the elsAxdt takes from the CGNS tree, refer to the section .

The load mode flags are dependant. This means some of them are forced if you ask for another one. For example READ_MESH is forced if you ask only for READ_CONNECT or READ_BC.

The load flag has to be set on the mode attribute:

```
xdt=XdtCGNS("file.cgns")
myflag=READ_MESH
xdt.mode=myflag
```

The way to unset a flag is to make a logical AND with the existing flags and a NOT of the flag you want to unset. In the example below, the trace mode is set to OFF.

```
xdt.mode &= ~elsAxdt.READ_TRACE
```

3.7

4. PROFILE

The logical organisation of data managed by elsAxdt is CGNS compliant. The Python trees you have as input, output or even available during run-time have a CGNS data model, the node names, types, their value meaning and the structure of node parents and children are those recommanded by CGNS.

4.1 Compliance and profile

The compliance to CGNS means that all data you will give as input to the solver are compliant, and all data you get from the solver are compliant too. This doesn't mean that elsa can read and write all CGNS data. An obvious example is unstructured data. The elsa solver doesn't read the unstructured grids, even if these grids are described in SIDS (the reference document for CGNS data model).

4.1.1 Trees and nodes

A tree is a node with children nodes. The first node is said to be the root and the nodes at the first level below the root are the children nodes. The CGNS logical structure is a tree, where nodes have a name, a type and may have a value.

For example, we want to represent the X coordinates for a mesh, we use a node with the name *`CoordinateX`*, its type is *`DataArray_t`* and it contains an array of float values representing these X coordinates. The node can be identified by its full name, i.e. the path *`/Base/Zone/GridCoordinates/CoordinateX`* which also indicates that the coordinates are related to a given zone in a given base. Most CGNS node names are the names of the value itself instead of the usual CFD variable. For example, *`SpecificHeatRatio`* is used in the standard instead of the usual *`Gamma`*.

4.1.2 The solver profile

Then we have to define the restriction of elsa to SIDS. In other words, what is the list of CGNS structure that can be understood by elsa. The mandatories structures should be there, but there are many optional structures and fixed choices we have to describe. All these specfication is also known as the solver profile. The restrictions and extensions of the elsa/CGNS data specification with respect to the CGNS standard are listed in the same order that the SIDS document:

- Specific non matching grid connectivities
- Extension for partial or extended fields data
- Specific boundary conditions
- Computing and numerical control attributes

The specific non matching connectivities have been added to take into some elsa extra capabilities.

The *Extensions for partial fields data* have been added in order to store 1D or 2D data in a 3D field for example. The CGNS standard defines how to organize the data associated with the whole zone fields, but there is no guidelines for the definition or partial fields results (such as a surface). The extension describes the way the elsa solver defines these partial fields.

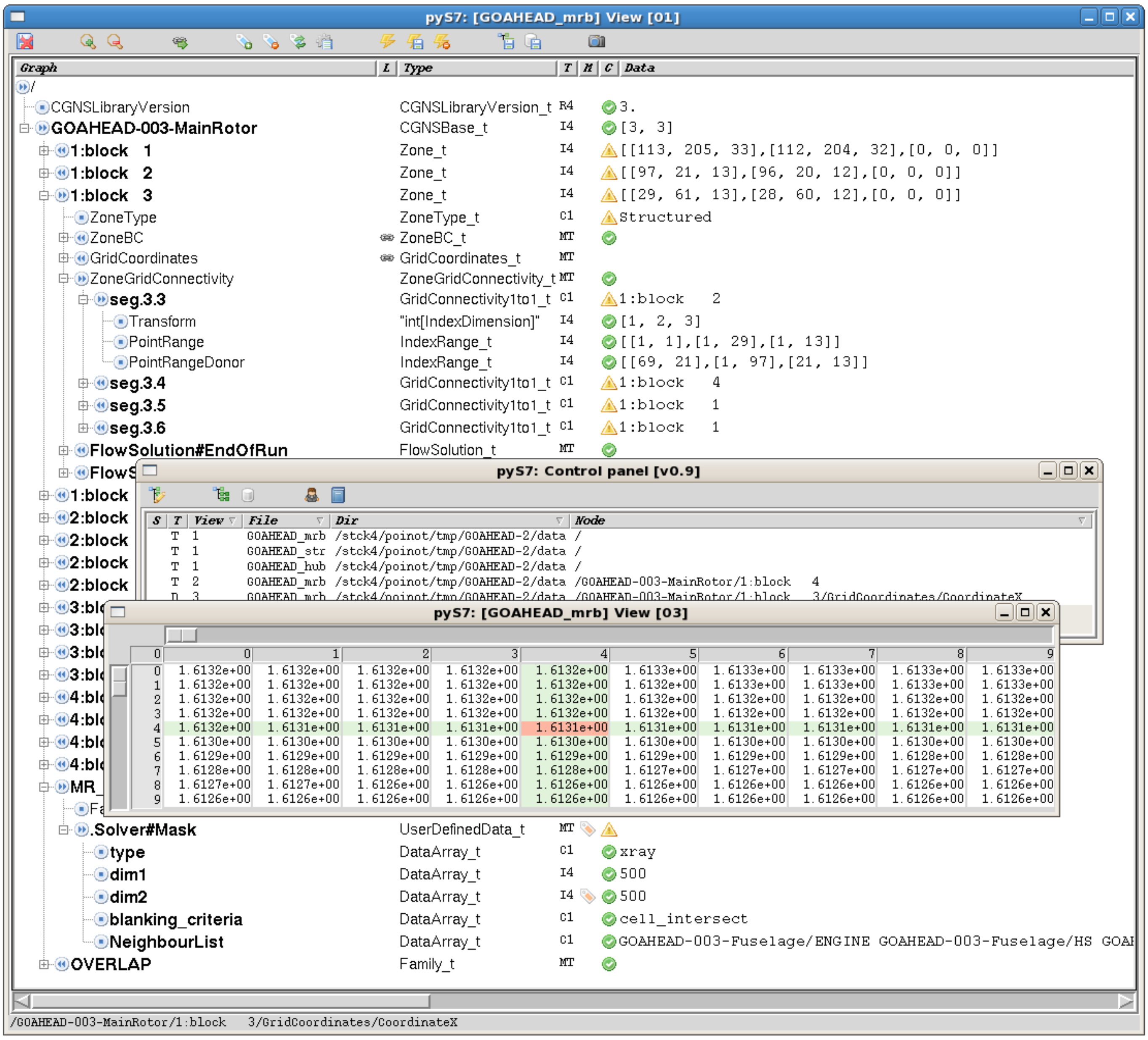
The *Specific boundary conditions* are the elsA boundary condition that have no matching types in the CGNS standard.

The *Computing and numerical control attributes* are all the elsa solver specific control values, such as number of iterations, multigrid cycles, artificial viscosity, or any other parameter related to the solver but not taken into account in the CGNS standard.

The elsA solver specific nodes are defined to store information dedicated to elsA, most of them have the `UserDefinedData_t` node type and they have the `.Solver#` prefix.

4.1.3 About screenshots

The CGNS trees screenshots have been produced by CGNS.NAV, which can read HDF5/ADF binary files or Python in-memory CGNS trees.



pyS7: [GOAHEAD_mrb] View [01]

Graph	L	Type	T	M	C	Data
/						
CGNSLibraryVersion		CGNSLibraryVersion_t	R4			3.
GOAHEAD-003-MainRotor		CGNSBase_t	I4			[3, 3]
1:block 1		Zone_t	I4			[[113, 205, 33],[112, 204, 32],[0, 0, 0]]
1:block 2		Zone_t	I4			[[97, 21, 13],[96, 20, 12],[0, 0, 0]]
1:block 3		Zone_t	I4			[[29, 61, 13],[28, 60, 12],[0, 0, 0]]
ZoneType		ZoneType_t	C1			Structured
ZoneBC		ZoneBC_t	MT			
GridCoordinates		GridCoordinates_t	MT			
ZoneGridConnectivity		ZoneGridConnectivity_t	MT			
seg.3.3		GridConnectivity1to1_t	C1			1:block 2
Transform		"int[IndexDimension]"	I4			[1, 2, 3]
PointRange		IndexRange_t	I4			[[1, 1],[1, 29],[1, 13]]
PointRangeDonor		IndexRange_t	I4			[[69, 21],[1, 97],[21, 13]]
seg.3.4		GridConnectivity1to1_t	C1			1:block 4
seg.3.5		GridConnectivity1to1_t	C1			1:block 1
seg.3.6		GridConnectivity1to1_t	C1			1:block 1
FlowSolution#EndOfRun		FlowSolution_t	MT			
FlowS						
1:block						
2:block						
2:block						
2:block						
2:block						
3:bl						
3:bl						
3:bl						
3:bl						
4:bl						
4:bl						
4:bl						
4:bl						
MR						
Fa						
.Solver#Mask		UserDefinedData_t	MT			
type		DataArray_t	C1			xray
dim1		DataArray_t	I4			500
dim2		DataArray_t	I4			500
blanking_criteria		DataArray_t	C1			cell_intersect
NeighbourList		DataArray_t	C1			GOAHEAD-003-Fuselage/ENGINE GOAHEAD-003-Fuselage/HS GOA
OVERLAP		Family_t	MT			

pyS7: Control panel [v0.9]

S	T	View	File	Dir	Node
	T	1	GOAHEAD_mrb	/stck4/poinot/tmp/GOAHEAD-2/data	/
	T	1	GOAHEAD_str	/stck4/poinot/tmp/GOAHEAD-2/data	/
	T	1	GOAHEAD_hub	/stck4/poinot/tmp/GOAHEAD-2/data	/
	T	2	GOAHEAD_mrb	/stck4/poinot/tmp/GOAHEAD-2/data	/GOAHEAD-003-MainRotor/1:block 4
	D	3	GOAHEAD_mrb	/stck4/poinot/tmp/GOAHEAD-2/data	/GOAHEAD-003-MainRotor/1:block 3/GridCoordinates/CoordinateX

pyS7: [GOAHEAD_mrb] View [03]

0	0	1	2	3	4	5	6	7	8	9
0	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6133e+00	1.6133e+00	1.6133e+00	1.6133e+00	1.6133e+00
1	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6133e+00	1.6133e+00	1.6133e+00	1.6133e+00
2	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00
3	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00	1.6132e+00
4	1.6132e+00	1.6131e+00	1.6131e+00	1.6131e+00	1.6131e+00	1.6131e+00	1.6131e+00	1.6131e+00	1.6131e+00	1.6131e+00
5	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00	1.6130e+00
6	1.6129e+00	1.6129e+00	1.6129e+00	1.6129e+00	1.6129e+00	1.6129e+00	1.6129e+00	1.6129e+00	1.6128e+00	1.6128e+00
7	1.6128e+00	1.6128e+00	1.6128e+00	1.6128e+00	1.6128e+00	1.6127e+00	1.6127e+00	1.6127e+00	1.6127e+00	1.6127e+00
8	1.6127e+00	1.6127e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00
9	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00	1.6126e+00

```
/GOAHEAD-003-MainRotor/1:block   3/GridCoordinates/CoordinateX
```

In this view, the columns name are displayed. Other screen shoots in this manual may no have the column names. However, the contents is obvious: the first column is the name of the node, the second textual column is the type of the node. In most cases, this pair (name, type) is enough to understand the tree structure. The user-defined names are in bold, while the other are mandatory or strongly recommanded names.

4.2 *Main CGNS tree structures*

The input data is a read-only tree. The input includes the geometry of the computation (grids, connectivities), the boundary conditions, the initialization fields, the reference state, the governing equations and the computation and numerical control attributes.

All the parameters and data required for a computation can be described in a single logical tree structure. This tree can actually be stored into several CGNS files with links which are parsed by *elsAxdt*.The standard data is defined using the SIDSrecommandation. The proprietary information is represented using the SIDS user defined structures.

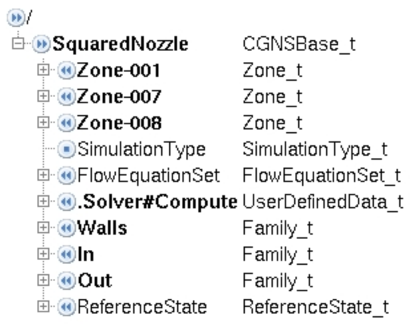
> △ All float values are double floats except the *CGNSLibraryVersion_t* value, which usually is reserved for internal use.
>
> △ The order of the children for a given node is not significant and has no effect on the parsing or the interpretation of data by the solver.

4.2.1 Common structures (SIDS 4 and 5)

The elsa solver performs its computation without any knowledge about the dimensionality of data. The *DataClass_t*, *DimensionalUnits_t*, *DimensionalExponents_t* (sections 4.1, 4.3 and 4.4) are unused. In the case such structures are found in the input tree, these are ignored. Thus, all data is understood as non dimensional data normalized with dimensional quantities. Thus, the *DataArray_t* pattern used as input as well as output doesn't have the dimensional informations and the *DataConversion_t* are ignored.

4.2.2 Entry level structures (SIDS 6.2)

The first node below the root node is the *Base_t* and it can have any user defined name. You can read as many CGNS trees you would like too, elsAxdt will add the CGNSBase_t to the previously loaded bases. In the case no base is found in the input tree, the error 7100 is raised. The *Base_t* defines the physical and topological dimensions of the whole data in the CGNS tree (a 3D base cannot contain 2D zones). As *elsAxdt* can have multiple CGNSBase_t, the path to the actual CGNS object should include the CGNSBase_t name. In the example below, you should refer to /SquaredNozzle/Zone-001 instead of Zone-001 (even if you have only one base).



```
/
SquaredNozzle      CGNSBase_t
  Zone-001         Zone_t
  Zone-007         Zone_t
  Zone-008         Zone_t
  SimulationType   SimulationType_t
  FlowEquationSet  FlowEquationSet_t
  .Solver#Compute  UserDefinedData_t
  Walls            Family_t
  In               Family_t
  Out              Family_t
  ReferenceState   ReferenceState_t
```

Fig (1) Base level children

The elsa solver ignores the *CellDimension* and *PhysicalDimension* attributes. The computation dimensions are set by the dimensions of the first zone found. So far, only 3D meshes can be read by elsa/CGNS. A 2D mesh has to be build with two indices in the K plane to obtain a 3D base, you have to do that with your own tools, then load the resulting 3D base with elsAxdt.

The *CGNSBase_t* node should have at least one *Zone_t*.

The single *ReferenceState_t* and the list of *Family_t* are taken into account.

The familly structure is stored by the elsAxdt CGNS tree parser, these families can be used for BCs or output commands features, these points are detailled hereafter in this document. You can see `Family_t` nodes are all at the Base level. Thus, one should take care of the families names.

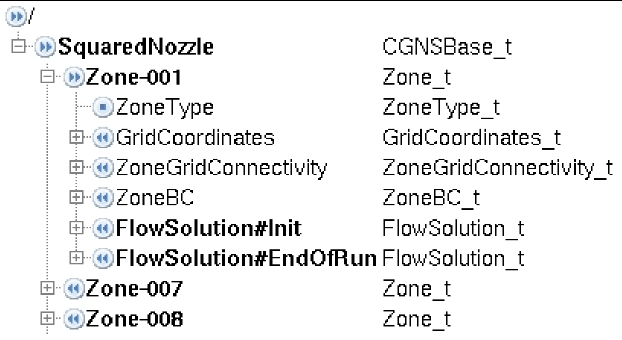
The user defined node `.Solver#Compute` should be found at the `CGNSBase_t` level. This node contains the solver specific parameters for the computation. The `FlowEquationSet_t` node also contains solver independant parameters.

All other nodes found as the `CGNSBase_t` children are ignored.

4.2.3 Zone structure (SIDS 6.3)

The physical discretization of space is separated in blocs or zones. The zones are children of the Base_t node and they have the Zone_t type. elsa only reads structured zones. The Zone_t nodes can have any user defined name. Below the zone node, we can find all grid-related information, boundary conditions, solutions and connectivities.

Connectivities are allowed only between zones of the same base. This means that the name of a zone refered-to in a connectivity structure should be found in the same base. This also means that if you do not have enough connectivity information to give to the solver (i.e. for a Chimera computation) you should not introduce a grid connectivity structure. In the case no zone is found in the input tree, the error 7101 is raised.

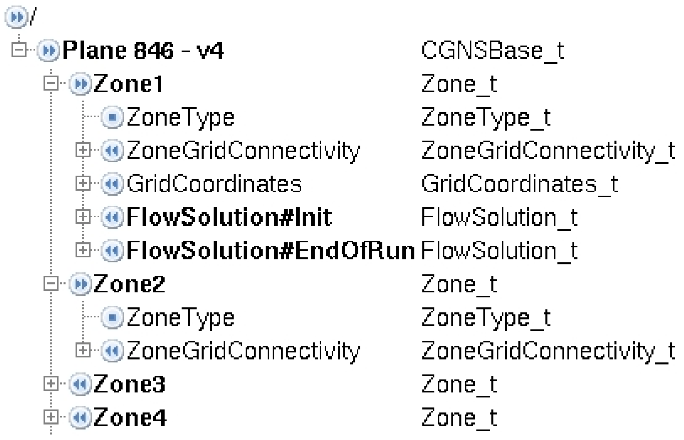


```
/
SquaredNozzle                        CGNSBase_t
  Zone-001                           Zone_t
    ZoneType                         ZoneType_t
    GridCoordinates                  GridCoordinates_t
    ZoneGridConnectivity             ZoneGridConnectivity_t
    ZoneBC                           ZoneBC_t
    FlowSolution#Init                FlowSolution_t
    FlowSolution#EndOfRun            FlowSolution_t
  Zone-007                           Zone_t
  Zone-008                           Zone_t
```

Fig (2) Zone example

The children of the `Zone_t` node are the grid itself (`GridCoordinates`), the connectivities between this zone and other zones (`ZoneGridConnectivity_t`), the boundary conditions (`ZoneBC_t`), the flow solutions (`FlowSolution_t`).

In the case of a parallel computation, the zones taken into account by the current processor should have a complete structure. The zones which are not located on the current processor may only contain the connectivity information.



```
/
Plane 846 - v4                       CGNSBase_t
  Zone1                              Zone_t
    ZoneType                         ZoneType_t
    ZoneGridConnectivity             ZoneGridConnectivity_t
    GridCoordinates                  GridCoordinates_t
    FlowSolution#Init                FlowSolution_t
    FlowSolution#EndOfRun            FlowSolution_t
  Zone2                              Zone_t
    ZoneType                         ZoneType_t
    ZoneGridConnectivity             ZoneGridConnectivity_t
  Zone3                              Zone_t
  Zone4                              Zone_t
```

Fig (3) Complete and incomplete zones (parallel computation)

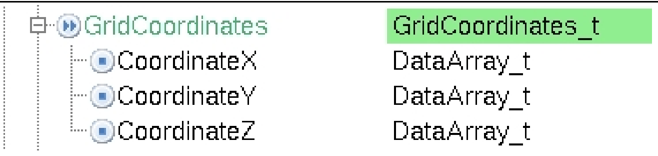
In the figure Fig (3) you can see the *Zone1* is complete, the *Zone2* is not, it only has the connectivity information. Such a tree can be given to the processor having in charge only the *Zone1*, assuming another processor would have *Zone2* to compute.

4.3 CGNS tree grids and solutions

The meshes and solutions are children of their associated zone. The solution one can found in a zone contains the full field of data related to the mesh. We detail here how to drive elsA for the output tree specification.

4.3.1 Grids (SIDS 7.1)

The CGNS mesh is defined by the sub-tree `GridCoordinates` containing the values for `CoordinateX`, `CoordinateY`, and `CoordinateZ`. These names are mandatory. The grids coordinates should be stored in a per-coordinate basis. The CGNS standard requires that any vector should be stored with a data array per vector coordinate, the array indexing should have the Fortran indexing (i.e.`KJI` loop).



```
GridCoordinates     GridCoordinates_t
  CoordinateX       DataArray_t
  CoordinateY       DataArray_t
  CoordinateZ       DataArray_t
```

Fig (4) Coordinate arrays for a 3D grid

△ Coordinates should be double precision data

△ The `Rind` information is not taken into account, neither in grids nor in solutions.

△ The grid coordinates are not duplicated in the output tree, unless explicitely requested by the user application as described in the next section.

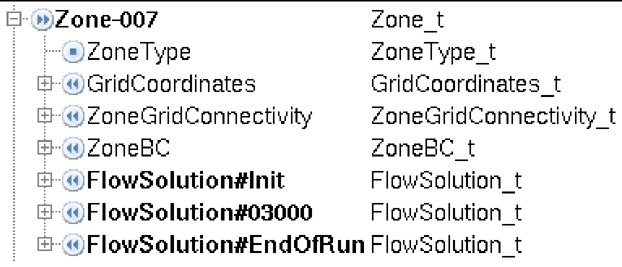
In the case of a moving grid, the pattern `Grid#<label>` is recommanded. The elsAxdt parser would only read the `GridCoordinates` node and it is up to the user application to associate links between the actual grid to use at a given step.

4.4 FlowSolution (SIDS 7.7)

The `FlowSolution_t` node is reserved for computation solution having the same size than the grid itself. The solution nodes should use the *elsAxdt* reserved names, at least for the initialisation and for the requested output solution.

The recommanded name pattern is `FlowSolution#<label>` with `<label>` the iteration number, `Init` or `EndOfRun`. There is only one reserved and mandatory name in the case of a restart, the initialization solution should have the name `FlowSolution#Init`.

The expected output solution in the input tree and the actual output solution in the output tree have a name with the pattern `FlowSolution#<label>`. When elsAxdt finds such a flow solution node in the input three, it generates the same skeletton for the output tree. This output tree will have a `FlowSolution_t` node with the name `FlowSolution#<label>`.



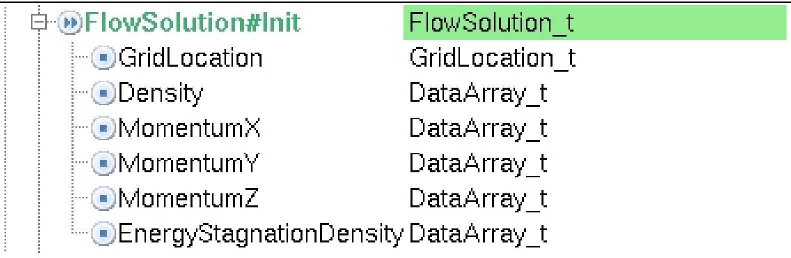
```
Zone-007                  Zone_t
  ZoneType                ZoneType_t
  GridCoordinates         GridCoordinates_t
  ZoneGridConnectivity    ZoneGridConnectivity_t
  ZoneBC                  ZoneBC_t
  FlowSolution#Init       FlowSolution_t
  FlowSolution#03000      FlowSolution_t
  FlowSolution#EndOfRun   FlowSolution_t
```

Fig (5) Flow solutions in a Zone

You can have as many `FlowSolution#<label>` solutions as you want. A good practice is to use `FlowSolution#<iteration>` for the output name and set a link from `FlowSolution#Init` to the initialisation solution.

A `FlowSolution_t` without the `FlowSolution#` prefix is ignored by elsAxdt.

The `FlowSolution#Init` would define the conservative variables at the cell center location. In the case elsAxdt finds more variable than required, the extra are ignored. Of there is less variable than expected an error is raised by the solver kernel. The distance to the wall required by the turbulence functions can also be set in the initialisation `FlowSolution_t`. The list of expected variables are detailled in the next table.



FlowSolution#Init	FlowSolution_t
GridLocation	GridLocation_t
Density	DataArray_t
MomentumX	DataArray_t
MomentumY	DataArray_t
MomentumZ	DataArray_t
EnergyStagnationDensity	DataArray_t

Fig (6) Flow solution used to initialize a Zone

△ The turbulence variables are declared using their actual name, not the variable name. For example, one should use `TurbulentEnergyKineticDensity` instead of the elsa `v6`.

The usual variable used in the init are:

- `Density, MomentumX, MomentumY, MomentumZ, EnergyStagnationDensity`
- `TurbulentSANuTildeDensity` (Spalart turbulence model only)
- `TurbulentEnergyKineticDensity, TurbulentDissipationDensity` (K turbulence models)
- `TurbulentDistance, TurbulentDistanceIndex` (only for distance computation)

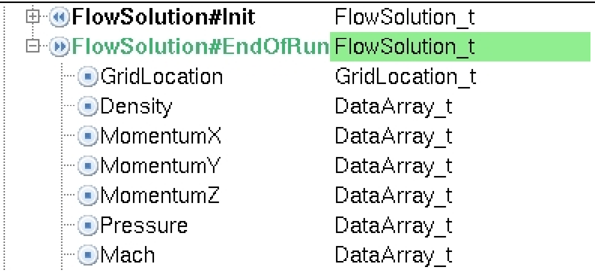
The can be set without conservative variables. You have four options for the initialization:

- `A ReferenceState` and no distance information (you may have to set it as a `.Solver#compute` attribute)
- A `ReferenceState` and the `TurbulentDistance, TurbulentDistanceIndex` in the `FlowSolution#Init`
- A `FlowSolution#Init` with conservative variables and no distance information (you may have to set it as a `.Solver#compute` attribute)
- A `FlowSolution#Init` with conservative variables and distance information.

4.4.1 Empty solution as ouput skeletton

Any solution with a name starting with `FlowSolution#EndOfRun` is understood as an output specification by elsAxdt. Such a solution should be found in each zone where the user wants an output.

When elsAxdt finds a `FlowSolution#EndOfRun` name, it generates an output skeletton for the computation. In other words, you give a skeletton of ouput tree and elsa uses it as the ouput tree model. The children of this FlowSolution_t node can of `DataArray_t` of `UserDefinedData_t` types. Only the name is used as the variable name to add to the output tree.



FlowSolution#Init	FlowSolution_t
FlowSolution#EndOfRun	FlowSolution_t
GridLocation	GridLocation_t
Density	DataArray_t
MomentumX	DataArray_t
MomentumY	DataArray_t
MomentumZ	DataArray_t
Pressure	DataArray_t
Mach	DataArray_t

Fig (7) Flow solution result a end of computation

△ If you define a `DataArray_t` you will have to set the zone dimensions in order to be CGNS compliant. Thus you will loose a lot of space if you do not lay on an existing solution you want to keep. Moreover, as using the `DataArray_t` does not require the zone dimensions, you can use a link to a given zone and then factorize your output declaration.

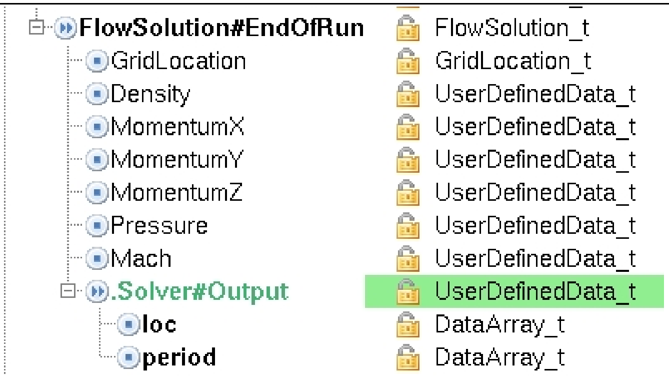
△ In the output tree, no links are created to the grids coordinates, reference state or any other sub-trees of the input CGNS tree. The user as to post-process the result and add the links to the grids.

△ As mentionned in the next section, a request to a conservative variable implicitely forces the output of the corresponding convergence.

△ As the init solution and the output solution may have the same dimensions, it is possible to use the init solution as output skeletton, again using the links mechanisms.

A special name is reserved, in the case your solution starts with *`FlowSolution#AbsoluteCoordinates`*, the parser would force the coordinates output to be absolute. These coordinates are put in the flow solution sub-node, not in the grid coordinates sub-node.

You can set the elsa specific parameters using the `.Solver#Output` node. You add a child with the elsa parameter name as node name and its value with a `DataArray_t` node type.



```
FlowSolution#EndOfRun        FlowSolution_t
    GridLocation             GridLocation_t
    Density                  UserDefinedData_t
    MomentumX                UserDefinedData_t
    MomentumY                UserDefinedData_t
    MomentumZ                UserDefinedData_t
    Pressure                 UserDefinedData_t
    Mach                     UserDefinedData_t
    .Solver#Output           UserDefinedData_t
        loc                  DataArray_t
        period               DataArray_t
```

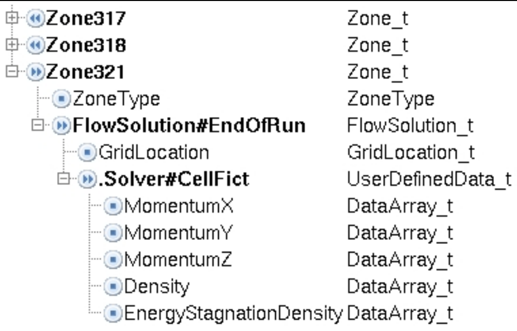
Fig (8) Proprietary settings for a solution output

For example, for a given flow solution, the grid location of the data is fixed. If you want `Vertex` solutions and a CellCenter solutions you should have two separate solutions, or per grid location. Use the loc attribute defined below. The default value is CellCenter.

The `.Solver#Output` node is a `UserDefinedData_t` node with the definition of some elsa output control parameters:

`period`	integer, sets output frequency w.r.t. iteration	refer to elsA manual
`writingmode`	integer 1: end of run 2: one each period	refer to elsA manual
`writingframe`	integer	refer to elsA manual
`loc`	integer 0: CellCenter 1: Vertex 2: cellfict	Sets the GridLocation for the expected output

Please note the important case of the so-called cellfict output . This output is not CGNS compliant and cannot be stored as a `FlowSolution_t`. Then we add a `UserDefinedData_t` node named `.Solver#CellFict` which stores the cellfict array values.



```
Zone317                          Zone_t
Zone318                          Zone_t
Zone321                          Zone_t
  ZoneType                       ZoneType
  FlowSolution#EndOfRun          FlowSolution_t
    GridLocation                 GridLocation_t
    .Solver#CellFict             UserDefinedData_t
      MomentumX                  DataArray_t
      MomentumY                  DataArray_t
      MomentumZ                  DataArray_t
      Density                    DataArray_t
      EnergyStagnationDensity    DataArray_t
```

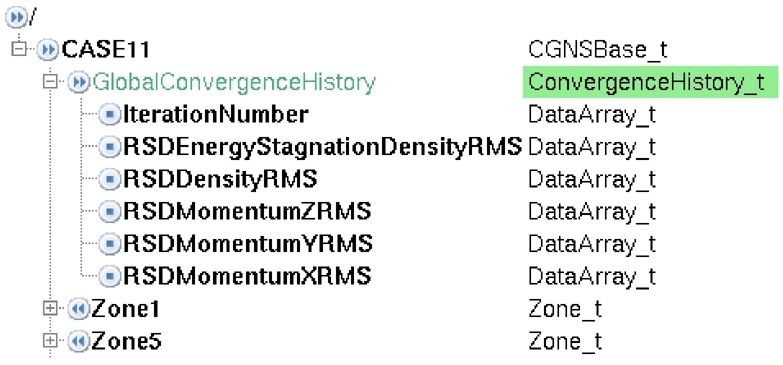
Fig (9) The proprietary so-called *CellFict* solution

The cellfict output will remain available for migration purpose, however, the user should note that in the future version of elsAxdt the storage of such a solution will be performed using a CGNS compliant structure.

The usual values used for output are detailled in the annex.

4.4.2 Convergence output

When you define a conservative variable in the solution skeletton, the convergence for this variable is automatically added in the output tree. The convergence would be produced at the `CGNSBase_t` level (`GlobalConvergenceHistory_t`).



```
/
CASE11                                CGNSBase_t
  GlobalConvergenceHistory            ConvergenceHistory_t
    IterationNumber                   DataArray_t
    RSDEnergyStagnationDensityRMS     DataArray_t
    RSDDensityRMS                     DataArray_t
    RSDMomentumZRMS                   DataArray_t
    RSDMomentumYRMS                   DataArray_t
    RSDMomentumXRMS                   DataArray_t
  Zone1                               Zone_t
  Zone5                               Zone_t
```

Fig (10) Global convergence for all Zones

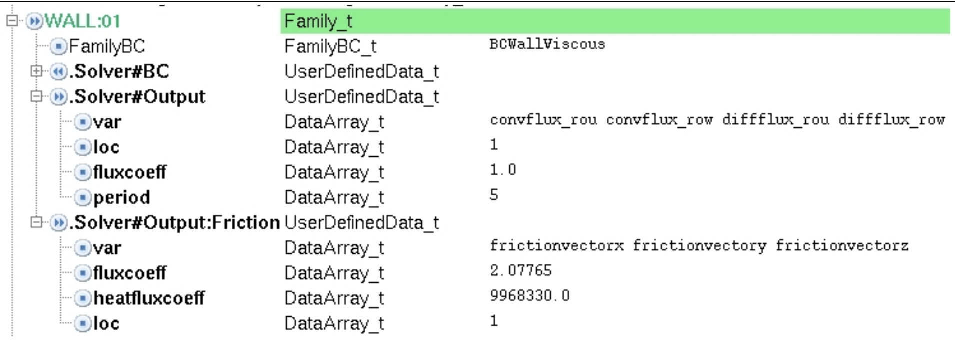
△ For parallel configurations, only the output tree on the processor 0 (zero) would have the convergence node. Again, it is up to the user application to gather parallel solution into a single tree. Usually, the post-processing of a CGNS tree in parallel performs the merge of the per-processor sub-trees and adds links to the input tree grid coordinates. With such a post-processing, a third-party vizualisation tool can blindly read your CGNS solution.

△ The window-based residual should be associated with a Family.

4.4.3 Family based output

The user can request convergence on other variables using a specific solver declaration with the families. A *`Family_t`* node has to have a *`.Solver#Output`* node with the set of elsa parameters for such an ouput.

The example in figure 4.12 shows a request for the family named *`Wall:01`* (which is also used to gather a set of BC windows, as explained later in this document in the boundary conditions section). The elsa parameters as defined as is, using `DataArray_t` node types to store float, integer or string values. The var is the space separated list of the elsa variable to use (e.g. convflux_rou, ...). Please refer to the elsa reference manual for the attribute values.



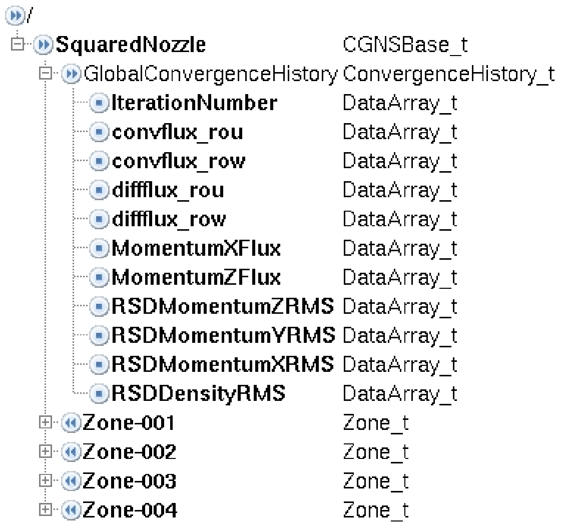
```
WALL:01                      Family_t
  FamilyBC                   FamilyBC_t        BCWallViscous
  .Solver#BC                 UserDefinedData_t
  .Solver#Output             UserDefinedData_t
    var                      DataArray_t       convflux_rou convflux_row diffflux_rou diffflux_row
    loc                      DataArray_t       1
    fluxcoeff                DataArray_t       1.0
    period                   DataArray_t       5
  .Solver#Output:Friction    UserDefinedData_t
    var                      DataArray_t       frictionvectorx frictionvectory frictionvectorz
    fluxcoeff                DataArray_t       2.07765
    heatfluxcoeff            DataArray_t       9968330.0
    loc                      DataArray_t       1
```

Fig (11) Output settings based on a family

△ *`<BaseName>/GlobalConvergenceHistory`* as path for usual convergence outputs.

△ The output specification node should have at least the pattern *`<BaseName>/<FamilyName>/.Solver#Output`* and it is possible to have more than one output specification.

The following figure shows an example of conservative variable convergence together with explicitely requested convergence. In the case of a global integration of the value, the result is set in `GlobalConvergenceHistory`.

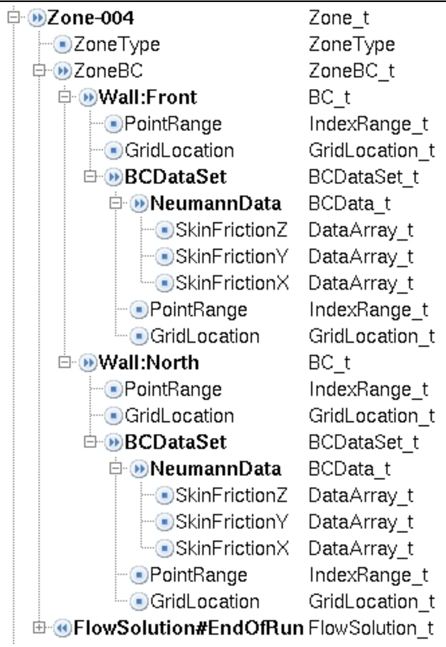


```
/
  SquaredNozzle                CGNSBase_t
    GlobalConvergenceHistory   ConvergenceHistory_t
      IterationNumber          DataArray_t
      convflux_rou             DataArray_t
      convflux_row             DataArray_t
      diffflux_rou             DataArray_t
      diffflux_row             DataArray_t
      MomentumXFlux            DataArray_t
      MomentumZFlux            DataArray_t
      RSDMomentumZRMS          DataArray_t
      RSDMomentumYRMS          DataArray_t
      RSDMomentumXRMS          DataArray_t
      RSDDensityRMS            DataArray_t
    Zone-001                   Zone_t
    Zone-002                   Zone_t
    Zone-003                   Zone_t
    Zone-004                   Zone_t
```

Fig (12) Convergence example

In the case of a partial field (i.e. a surface on a wall), the result goes into the related boundary condition, if you have a set of BCs, each BC has its own data. The Fig (13) is the output you obtain when you have Fig (11) as input.

For example, you have a set of `BC_t` nodes defining a surface and you want to have the pressure on this surface. You can create a `Family_t` named `SURFACE`, all related `BC_t` are `FamilySpecified` with `SURFACE` as `FamilyName`. In the `Family_t` node, at the `CGNSBase_t` level, you add the `.Solver#Ouput` reqesting the pressure output on this family. elsAxdt parses all the corresponding Bcs and adds an output for each.



```
Zone-004                          Zone_t
  ZoneType                        ZoneType
  ZoneBC                          ZoneBC_t
    Wall:Front                    BC_t
      PointRange                  IndexRange_t
      GridLocation                GridLocation_t
      BCDataSet                   BCDataSet_t
        NeumannData               BCData_t
          SkinFrictionZ           DataArray_t
          SkinFrictionY           DataArray_t
          SkinFrictionX           DataArray_t
        PointRange                IndexRange_t
        GridLocation              GridLocation_t
    Wall:North                    BC_t
      PointRange                  IndexRange_t
      GridLocation                GridLocation_t
      BCDataSet                   BCDataSet_t
        NeumannData               BCData_t
          SkinFrictionZ           DataArray_t
          SkinFrictionY           DataArray_t
          SkinFrictionX           DataArray_t
        PointRange                IndexRange_t
        GridLocation              GridLocation_t
  FlowSolution#EndOfRun           FlowSolution_t
```

Fig (13) Surface output

Such a BC based partial solution storage is used for code coupling purpose. The next section describes how to define such output.

4.4.3.a- Partial solutions stored as BC data

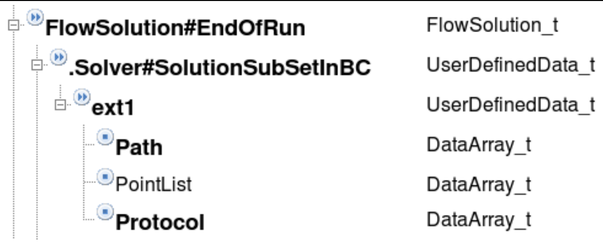
The partial field exchange can be used in order to retrieve a partial field such as a surface, but it is also designed for code-coupling data exchange. One should take care of the input tree declaration, which defines a structure in the solution sub-tree, while the result itself is found in the `ZoneBC_t` sub-tree.

> △ We are waiting for CGNS extensions so-called *Regions*, that would allow to store the partial fields in a flow solution still being compliant with CGNS SIDS.

The definition of the requested ouput is a private node *`.Solver#SolutionSubSetInBC`* under the *`FlowSolution#EndOfRun`*. This private node contains a list of *`UserDefinedData_t`* nodes each describing an expected output. The output is described by:

- ☐ *`PointList`* or *`PointRange`* the indice of the partial field in the current zone.
- ☐ *`Path`* a list of strings giving the name of the output value, w.r.t. the BC data sets structures.
- ☐ *`Protocol`* in the case of code coupling, defines the states of the automata where the input/output should occur. It is a list of strings, each string is a state keyword (*`[before, begin, iteration, subiteration, end, after]`*.

> △ elsa only reads structured zones, however, all ranges can be defined using `PointRange` of `PointList` structures. For example, it makes possible the use of both structured and unstructured codes at the boundaries if you define you point ranges using `PointList` instead of PointRange_t structure.



```
FlowSolution#EndOfRun             FlowSolution_t
  .Solver#SolutionSubSetInBC      UserDefinedData_t
    ext1                          UserDefinedData_t
      Path                        DataArray_t
      PointList                   DataArray_t
      Protocol                    DataArray_t
```

Fig (14) Surface output settings

For example, in `ext1` the `Path` could be:

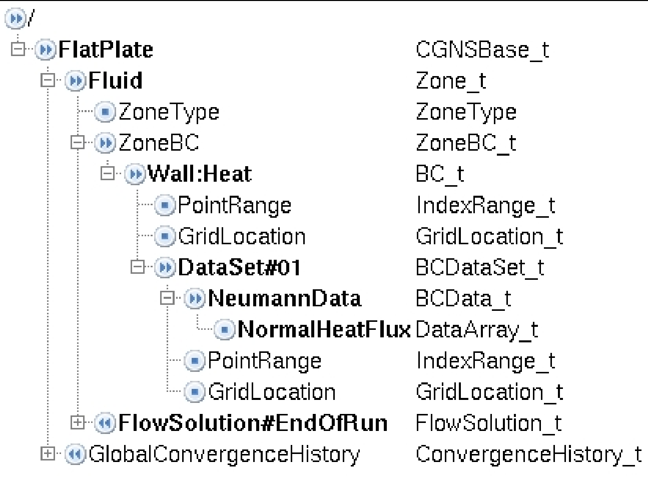
```
IcingWallData:Gaz/NeumannData/NormalHeatFlux
IcingWallData:Gaz/NeumannData/SkinFrictionY
IcingWallData:Gaz/NeumannData/SkinFrictionZ
```

and the Protocol could be after (end of computation). Such a definition means that we want the `NormalHeatFlux` and the `SkinFrictionY` and `SkinFrictionZ` on the specified `PointList` of the zone, at the end of the computation.

The `Path` strings should be separeted by newlines (`\n`), all trailing blanks are removed by the elsAxdt parser. A if variable name is not allowed for such an output, elsAxdt ignores the entry.

> △ In the current elsAxdt versions, the `PointList` is always interpreted as a `PointRange`.

See next section for details on where the partial field will be located in the output CGNS tree.



```
/
FlatPlate                          CGNSBase_t
  Fluid                            Zone_t
    ZoneType                       ZoneType
    ZoneBC                         ZoneBC_t
      Wall:Heat                    BC_t
        PointRange                 IndexRange_t
        GridLocation               GridLocation_t
        DataSet#01                 BCDataSet_t
          NeumannData              BCData_t
            NormalHeatFlux         DataArray_t
          PointRange               IndexRange_t
          GridLocation             GridLocation_t
    FlowSolution#EndOfRun          FlowSolution_t
  GlobalConvergenceHistory         ConvergenceHistory_t
```

Fig (15) Surface output result

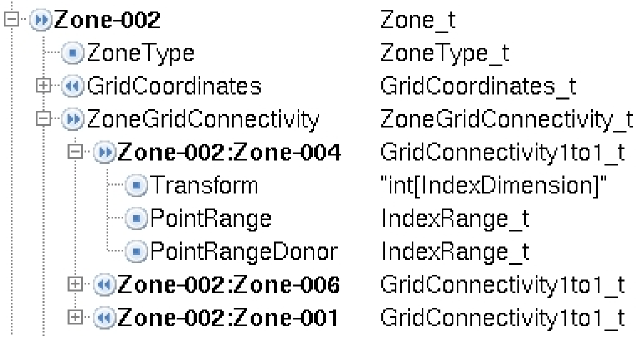
4.5 Grid connectivity (SIDS 8.1)

The mesh connectivity should be a multizone connectivity in the `ZoneGridConnectivity_t` node. The nodes taken into account are of type `GridConnectivity1to1_t` and `GridConnectivity_t` .

4.5.1 Matching connectivity

The connectivity with type `GridConnectivity1to1_t` is reserved to zone interface with matching patches. The connectivity sub-tree should declare one interface per zone. Then if you have a zone with more than one interface to another zone, you have to define one connectivity per patch. The onor zone, that is the opposite zone of the patch, would also have the definition of the interface.

Each sub-tree contains a local patch (*`PointRange`* or *`PointList`*) and the other zone patch (*`PointRangeDonor`* or *`PointListDonor`*). The other zone is identified by the *`ZoneDonorName`*. The Transform node gives the transformation matrix.



```
Zone-002                       Zone_t
  ZoneType                     ZoneType_t
  GridCoordinates              GridCoordinates_t
  ZoneGridConnectivity         ZoneGridConnectivity_t
    Zone-002:Zone-004          GridConnectivity1to1_t
      Transform                "int[IndexDimension]"
      PointRange               IndexRange_t
      PointRangeDonor          IndexRange_t
    Zone-002:Zone-006          GridConnectivity1to1_t
    Zone-002:Zone-001          GridConnectivity1to1_t
```

Fig (16) One-to-one connectivity

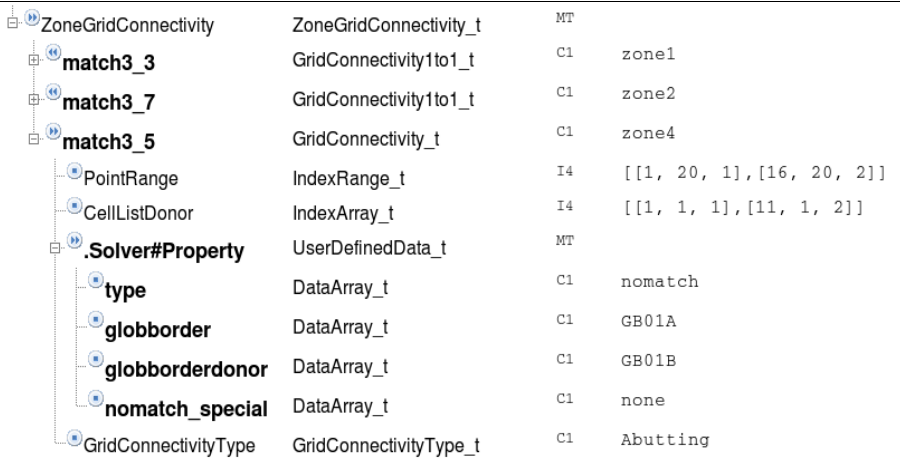
△ No *PointXxxx* and *PointXxxxDonor* mix with *Range* and

△ Ranges are list of indices by points, i.e. in 3D we have *[Imin, Jmin, Kmin, Imax, Jmax, Kmax]*. This allow the use of the same structure for 2D and for unstructured list of points indices.

4.5.1.a- Non-matching connectivity

The mismatch connectivity is described in a `GridConnectivity_t` node. The elsa solver requires the declaration of *GlobBorders* for such a connectivity. The connectivity is defined even if you do not have information about the connected zone(s), the `CellListDonor` is not used and the connected-to zone information is ignored too. The mandatory sub-node *.Solver#Property* of type `UserDefinedData_t` contains the information about the GlobBorders, these are:

type	*nomatch*	`Descriptor_t` or `DataArray_t`
globborder	string value	`Descriptor_t` or `DataArray_t`
globborderdonor	string value	`Descriptor_t` or `DataArray_t`
<any other attribute>	any value	`DataArray_t`



```
ZoneGridConnectivity      ZoneGridConnectivity_t   MT
  match3_3                GridConnectivity1to1_t   C1   zone1
  match3_7                GridConnectivity1to1_t   C1   zone2
  match3_5                GridConnectivity_t       C1   zone4
    PointRange            IndexRange_t             I4   [[1, 20, 1],[16, 20, 2]]
    CellListDonor         IndexArray_t             I4   [[1, 1, 1],[11, 1, 2]]
    .Solver#Property      UserDefinedData_t        MT
      type                DataArray_t              C1   nomatch
      globborder          DataArray_t              C1   GB01A
      globborderdonor     DataArray_t              C1   GB01B
      nomatch_special     DataArray_t              C1   none
    GridConnectivityType  GridConnectivityType_t   C1   Abutting
```

Fig (17) Proprietary so-called *nomatch* connectivity

Some defaults values are set by elsAxdt, you can always overwrite these values using attributes at the same node level.

△ The `CellListDonor` has to be present but can have fake values and should be of size 3x2, but in all cases, elsAxdt ignores it.

4.6 CGNS tree boundary conditions

The boundary conditions defined in CGNS are very general BCs. The elsa solver has a lot of dedicated BC and most descriptions in this section are extensions to the standard. Of course, these extensions are taking place in the standard itself and all presented CGNS trees are CGNS compliant. The elsa solver also uses the BC structure for a very specific purpose, the input of partial solution fields.

The elsa profile ZoneBC_t (SIDS 9.2) structure only supports `BC_t` children.

4.6.1 Boundary conditions (SIDS 9.3)

The elsa profile `BC_t` structure only supports `BCType_t`, `PointRange` and `PointList`, and `BCDataSet_t` children. The location of values is ignored at this level (see `BCDataSet_t`).

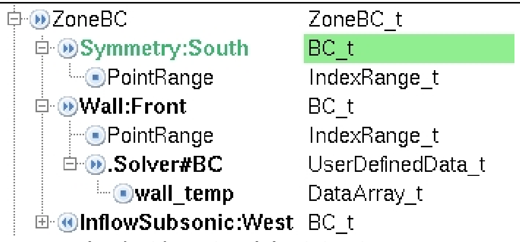
4.6.1.a- Boundary without data

The most simple BC definition is a simple type without data.

The *BC_t* contains a value, a string, which contains the CGNS BC type name. The actual name of the BC is left to the user. For example, you can find a BC with the name Wall:Up and with the value *BCWallInviscid*. The *PointRange* is the application patch or window for this BC. A *PointList* can replace the *PointRange*, as far as the list defines a structured patch.

The CGNS BC types without values and taken into account by elsa are listed hereafter. All other CGNS BC raises the error 7120. In the following table, the legacy elsa type is given for migration purpose.

BCFarfield	*nref*	
BCWallInviscid	*wallslip*	Normal velocity is ignored
BCWallViscous *BCWall*	*walladia*	
BCOutflowSupersonic	*outsup*	
BCSymmetryPlane	*sym*	
BCDegenerateLine	*inactive*	
BCDegeneratePoint	*inactive*	
UserDefined	*any*	Can be used to defined any elsA BC, mandatory for overlap BC definition.
BCOverlap	*overlap*	The use of this value is not recommanded, see Chimera section.



```
ZoneBC                      ZoneBC_t
  Symmetry:South            BC_t
    PointRange              IndexRange_t
  Wall:Front                BC_t
    PointRange              IndexRange_t
    .Solver#BC              UserDefinedData_t
      wall_temp             DataArray_t
  InflowSubsonic:West       BC_t
```

Fig (18) BC with CGNS type

The user defined BC indicates an elsa BC, with or without data, described in the next section.

△ In the case of a *PointList*, the min and the max of the point indices will be used, simulating a structured window. The *PointList* is only allowed in order to make life easier for connections with unstructured solvers.

△ Some CGNS BCs without arguments are translated in elsa BCs with solver-specific arguments. For example, *wallslip* can have the two *extrap* and *prescor* arguments. In that case, the user can either (1) supress these arguments from his CGNS tree, and the solver will use its own default values, or (2) use a *.Solver#BC* node as described hereafter. In the *wallslip* example, one would find two extra nodes, *.Solver#BC/extrap* and *.Solver#BC/prescor* which are integer *DataArray*s with a dimension of 1.

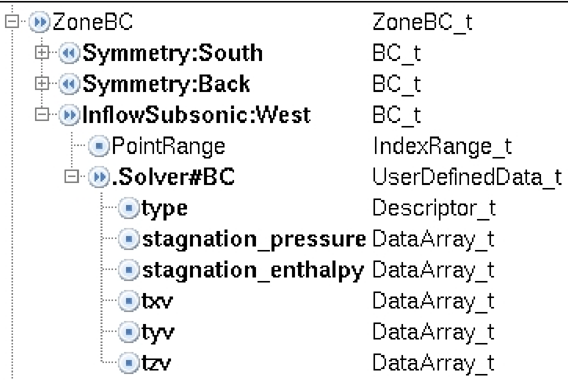
If the BC type does not exist in CGNS, or if it does not fit exactly with the elsa BC, you should use a complete *.Solver#BC* node. In that case, the BC type is *UserDefined* and the type is one of the arguments given in the BC pattern.

4.6.1.b- Boundary with global data

In that case, a list of values can be associated with the CGNS BC. This data, so-called DataSet, is defined without any knowledge of the application patch. This means the values are global to the whole application patch and is independant of any re-meshing. Most of the BC pattern is the same as the BC without data pattern. We add a DataSet subtree starting with a BCData_t node. The name of this node is left to the user and has no meaning to the solver. Then, depending on the kind of BC you have, you find a Dirichlet or Neumann node and the list of CGNS variables to define.

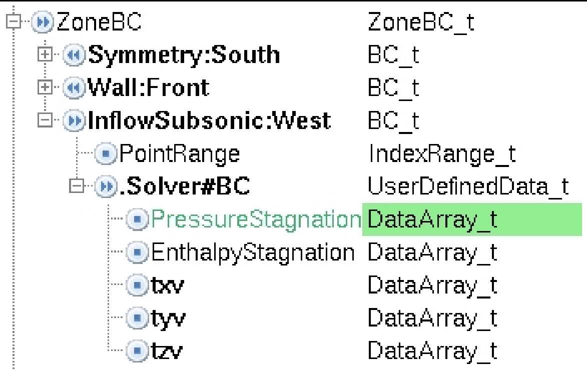
The CGNS BCs with global data accepted by elsa are listed in the table hereafter.

`BCInflowSupersonic`	`insup`	Requires the conservatives variables.
`BCInflowSubsonic` `BCTunnelInflow`	`inj1`	
`BCOutflowSubsonic` `BCTunnelOutflow`	`outpres`	
`BCWallViscousIsothermal`	*`wallisoth`*	
`BCWallViscousHeatFlux`	*`wallheatflux`*	



```
ZoneBC                          ZoneBC_t
  Symmetry:South                BC_t
  Symmetry:Back                 BC_t
  InflowSubsonic:West           BC_t
    PointRange                  IndexRange_t
    .Solver#BC                  UserDefinedData_t
      type                      Descriptor_t
      stagnation_pressure       DataArray_t
      stagnation_enthalpy       DataArray_t
      txv                       DataArray_t
      tyv                       DataArray_t
      tzv                       DataArray_t
```

Fig (19) Proprietary BC data



```
ZoneBC                          ZoneBC_t
  Symmetry:South                BC_t
  Wall:Front                    BC_t
  InflowSubsonic:West           BC_t
    PointRange                  IndexRange_t
    .Solver#BC                  UserDefinedData_t
      PressureStagnation        DataArray_t
      EnthalpyStagnation        DataArray_t
      txv                       DataArray_t
      tyv                       DataArray_t
      tzv                       DataArray_t
```

Fig (20) Proprietary BC data with CGNS variable names

△ It is always possible to add solver-specific values as described in the previous section. A .*`Solver#BC`* node has to be defined and the user can put a list of names and values.

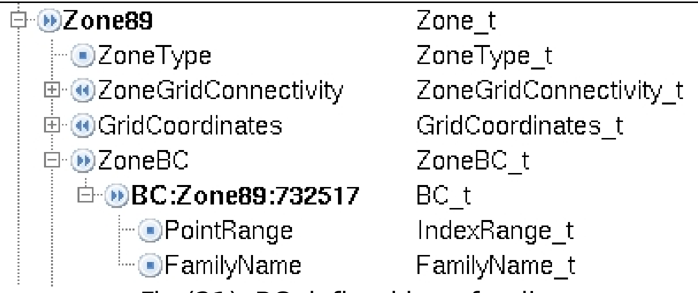
△ The conservative variables should have the \CGNS compliant name and should of type *`DataArray`* of float with a dimension of 1.

4.6.1.c- Family boundary conditions

The most user friendly way to define a boundary condition is to use the family logical set. Some tools can generate BC using the families. A Family is a label identification of a CAD surface, segment or point. The CAD tool allows the definition of BCs on families, thus allowing the use of set of CAD entities. As the CAD generates the mesh, is produces blocs, connectivities, distribute BCs and generates BCs' application patches.

The families in a CGNS tree can be found as BCs labels. This information is useless to the solver, as the

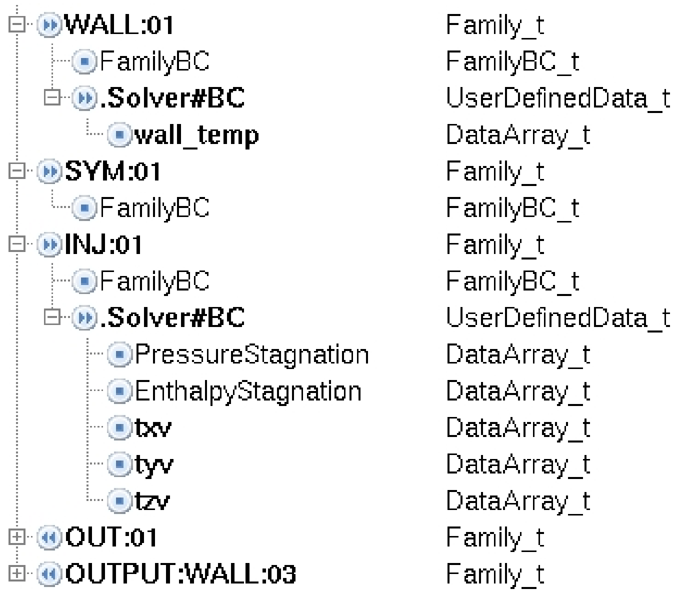
CAD distributes and correctly generates BC types, patches or global data. However, identification of families can be useful for pre and post processing, which is not the purpose of the elsa solver.



```
Zone89                          Zone_t
  ZoneType                      ZoneType_t
  ZoneGridConnectivity          ZoneGridConnectivity_t
  GridCoordinates               GridCoordinates_t
  ZoneBC                        ZoneBC_t
    BC:Zone89:732517            BC_t
      PointRange                IndexRange_t
      FamilyName                FamilyName_t
```

Fig (21) BC defined by a family

We can see the family gathers a set of BCs, each BC declares its application range in its zone and is associated to a family. The family itself is found in the base level, with the parameters to set for all the BCs that belongs to the family.



```
WALL:01                         Family_t
  FamilyBC                      FamilyBC_t
  .Solver#BC                    UserDefinedData_t
    wall_temp                   DataArray_t
SYM:01                          Family_t
  FamilyBC                      FamilyBC_t
INJ:01                          Family_t
  FamilyBC                      FamilyBC_t
  .Solver#BC                    UserDefinedData_t
    PressureStagnation          DataArray_t
    EnthalpyStagnation          DataArray_t
    txv                         DataArray_t
    tyv                         DataArray_t
    tzv                         DataArray_t
OUT:01                          Family_t
OUTPUT:WALL:03                  Family_t
```

Fig (22): Family BC and proprietary parameters

4.6.2 Partial fields

The partial fields are stored in BC structures. So far, the applications we have with partial fields are related to surfaces and can be understood as BC values. In the example below, the results have been defined by the skeletton found in FlowSolution_t/.Solver\#SolutionSetInBC} as described in a section before.

A CGNS BC with data set is created, with the compliant sub-tree to the actual value. The BC data set is in its owner zone and has a *PointRange* or *PointList* and a *GridLocation* defining the location of the values in its zone.

```
/Base/Zone/ZoneBC/Wall#1/IcingWallData:Gaz/NeumannData/NormalHeatFlux
/Base/Zone/ZoneBC/Wall#1/IcingWallData:Gaz/NeumannData/SkinFrictionY
/Base/Zone/ZoneBC/Wall#1/IcingWallData:Gaz/NeumannData/SkinFrictionZ
/Base/Zone/ZoneBC/Wall#2/IcingWallData:Gaz/NeumannData/SkinFrictionY
/Base/Zone/ZoneBC/Wall#2/IcingWallData:Gaz/NeumannData/SkinFrictionZ
```

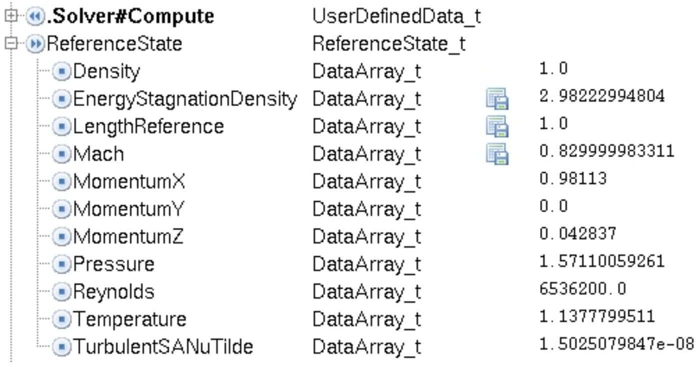
△ So far there is no control to check if the BC patch actually is on a surface or not.

△ There is no indication to tell wether the BC values are input or output values.

4.7 CGNS tree miscellaneous structures

There are CGNS sub-trees not related to the zones but rather to a global view of the computation.

4.7.1 The reference state (SIDS 12.1)



```
.Solver#Compute            UserDefinedData_t
ReferenceState             ReferenceState_t
  Density                  DataArray_t        1.0
  EnergyStagnationDensity  DataArray_t        2.98222994804
  LengthReference          DataArray_t        1.0
  Mach                     DataArray_t        0.829999983311
  MomentumX                DataArray_t        0.98113
  MomentumY                DataArray_t        0.0
  MomentumZ                DataArray_t        0.042837
  Pressure                 DataArray_t        1.57110059261
  Reynolds                 DataArray_t        6536200.0
  Temperature              DataArray_t        1.1377799511
  TurbulentSANuTilde       DataArray_t        1.5025079847e-08
```

Fig (23) Reference State data

Only one reference state named ReferenceState is allowed at the CGNSBase_t level. No other ReferenceState would be taken into account by the solver.

The `ReferenceState` should contain the relevant data with respect to the computation equation model. The table below details the possible patterns:

`Density`	Always required
`MomentumX`	Always required
`MomentumY`	Always required
`MomentumZ`	Always required
`EnergyStagnationDensity`	Always required
`TurbulentEnergyKineticDensity`	
`TurbulentDissipationDensity`	
`TurbulentSANuTilde`	
`TurbulentDistance`	
`TurbulentDistanceIndex`	

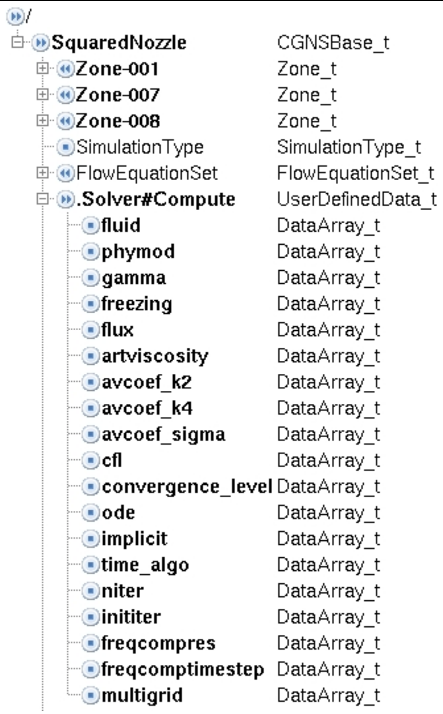
4.7.2 The flow equation set (SIDS 12.1)

The *`FlowEquationSet`* tree is ignored.

4.7.3 Solver computation parameters

We also added a proprietary sub-tree, required for the representation of all computation parameters used by elsa and not taken into account by the CGNS standard. These are solver parameters and CGNS could not specify these parameters as common for all CFD solvers. However, CGNS allows the addition of specific sub-trees such as our *`.Solver#Compute`* sub-tree).

The *`.Solver#Compute`* node contains all numerical control of the \elsa solver, the user usually cannot avoid to have this sub-tree.



```
/
SquaredNozzle             CGNSBase_t
  Zone-001                Zone_t
  Zone-007                Zone_t
  Zone-008                Zone_t
  SimulationType          SimulationType_t
  FlowEquationSet         FlowEquationSet_t
  .Solver#Compute         UserDefinedData_t
    fluid                 DataArray_t
    phymod                DataArray_t
    gamma                 DataArray_t
    freezing              DataArray_t
    flux                  DataArray_t
    artviscosity          DataArray_t
    avcoef_k2             DataArray_t
    avcoef_k4             DataArray_t
    avcoef_sigma          DataArray_t
    cfl                   DataArray_t
    convergence_level     DataArray_t
    ode                   DataArray_t
    implicit              DataArray_t
    time_algo             DataArray_t
    niter                 DataArray_t
    inititer              DataArray_t
    freqcompres           DataArray_t
    freqcomptimestep      DataArray_t
    multigrid             DataArray_t
```

Fig (24) Proprietary solver parameters

Again, the reader would refer to the tables to find the list and the meaning of every parameter.

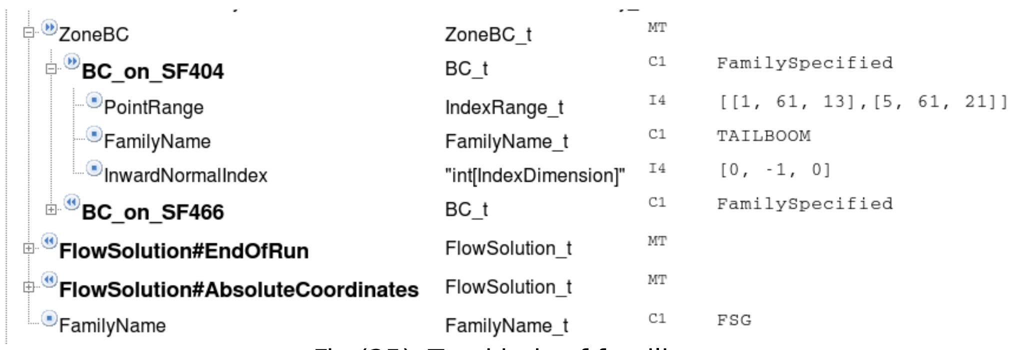
4.8 Chimera features

The Chimera requires the definition of the overlapping boundary conditions and the definition of the neighbourhood for a set of zones. A body is automatically detected in the case of a family BC defining a contiguous set of surfaces. This family is used as the tag for the body identification. A mask is created for this body. The zones without a mask have an automatic generation of ghost mask.

> △ The CGNS standard defines overlapping connectivity, in our case we are generating the overlapping data, thus the standard structure is unused.
>
> △ The read /write of overlapping information and Chimera coeffs will be availble in the next elsAxdt release.

The definition of the overlapping and the masks are using the families. The mask are Zone-level families and the overlapping are BC-level families. These should be different families, it is not possible to have the same family name for Zone and BC levels at the same time.

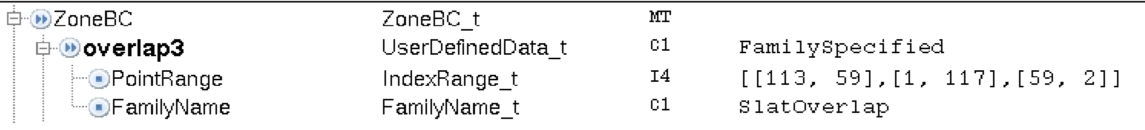
We see these two levels in the figure below, TAILBOOM defines a surface for the overlapping, while FSG is a family that gathers all the zones for the body and mask definition.



```
ZoneBC                                   ZoneBC_t                 MT
  BC_on_SF404                            BC_t                     C1   FamilySpecified
    PointRange                           IndexRange_t             I4   [[1, 61, 13],[5, 61, 21]]
    FamilyName                           FamilyName_t             C1   TAILBOOM
    InwardNormalIndex                    "int[IndexDimension]"    I4   [0, -1, 0]
  BC_on_SF466                            BC_t                     C1   FamilySpecified
FlowSolution#EndOfRun                    FlowSolution_t           MT
FlowSolution#AbsoluteCoordinates         FlowSolution_t           MT
FamilyName                               FamilyName_t             C1   FSG
```

Fig (25) Two kinds of families

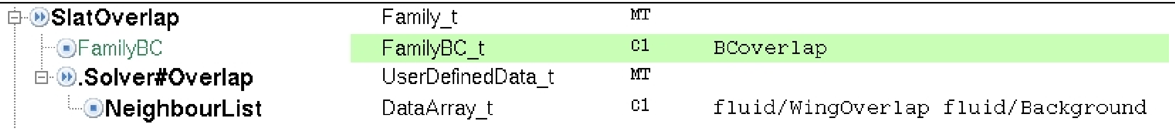
The overlap BC can be defined in two ways, either at the `BC_t` level or at the family level. We strongly recommand to use the family, which would be re-used for the neighbourhood:



ZoneBC	ZoneBC_t	MT	
overlap3	UserDefinedData_t	C1	FamilySpecified
PointRange	IndexRange_t	I4	[[113, 59],[1, 117],[59, 2]]
FamilyName	FamilyName_t	C1	SlatOverlap

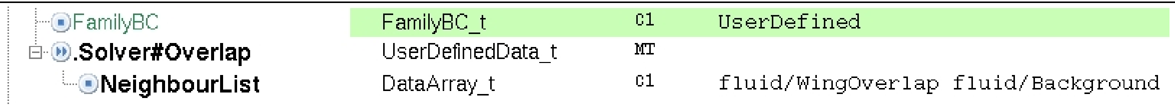
Fig (26) Family specified overlapping BC

The actual type of the BC can be `BCoverlap` (which is NOT a CGNS BC type) or `UserDefinedData`. We recommand the use of `UserDefinedData` which is standard. The `.Solver#Overlap` is a sub-node of this family, it defined the list of families used during neighbourhood search. The list is composed of `<CGNSBase_t name>/<Family_t name>` separated with space characters.



SlatOverlap	Family_t	MT	
FamilyBC	FamilyBC_t	C1	BCoverlap
.Solver#Overlap	UserDefinedData_t	MT	
NeighbourList	DataArray_t	C1	fluid/WingOverlap fluid/Background

Fig (27) Typed BC overlap definition

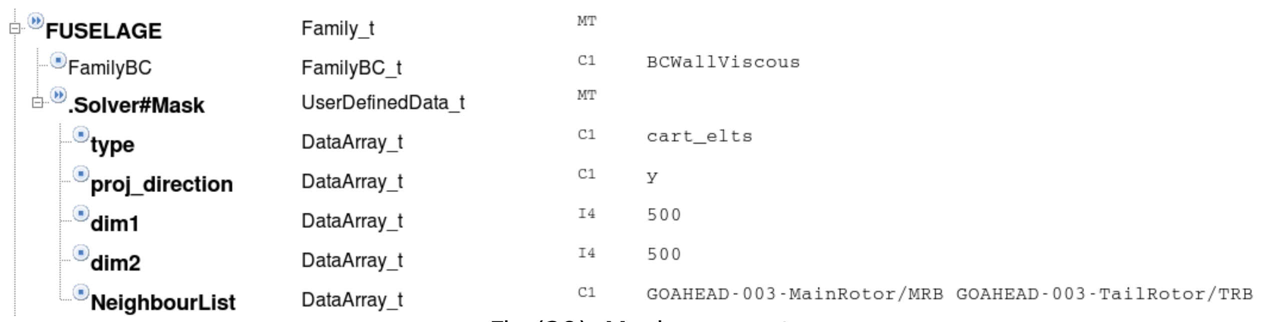


FamilyBC	FamilyBC_t	C1	UserDefined
.Solver#Overlap	UserDefinedData_t	MT	
NeighbourList	DataArray_t	C1	fluid/WingOverlap fluid/Background

Fig (28) UserDefined BC overlap definition

△ Note the the BC/FamilyName does NOT contain the `CGNSBase_t` name.

The specific parameters you want to set for the mask have to be set at the family level, with the `.Solver#Mask` node. The elsA attributes are used as-is, the `NeighbourList` has the same pattern as the previously described node.



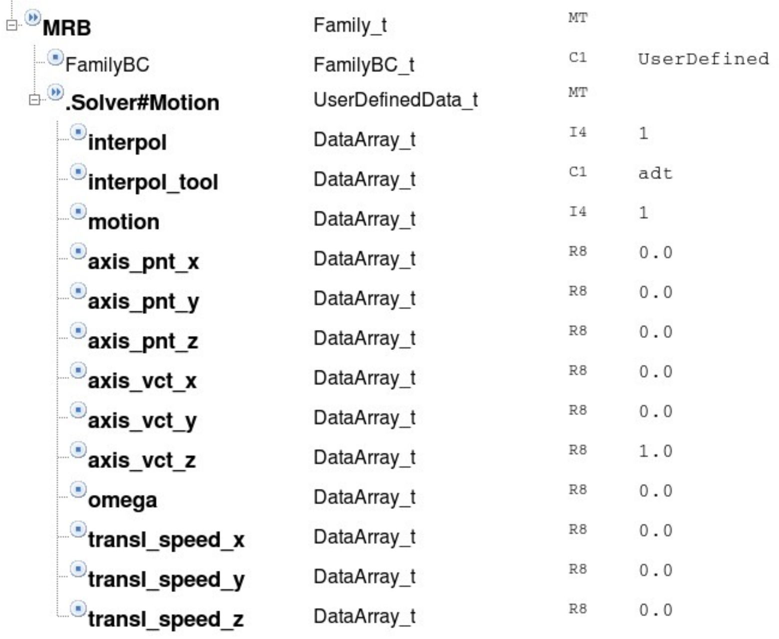
FUSELAGE	Family_t	MT	
FamilyBC	FamilyBC_t	C1	BCWallViscous
.Solver#Mask	UserDefinedData_t	MT	
type	DataArray_t	C1	cart_elts
proj_direction	DataArray_t	C1	y
dim1	DataArray_t	I4	500
dim2	DataArray_t	I4	500
NeighbourList	DataArray_t	C1	GOAHEAD-003-MainRotor/MRB GOAHEAD-003-TailRotor/TRB

Fig (29) Mask parameters

4.9 Zone parameters

The are a lot of elsA parameters the user can set at the Zone level. Most of the time the parameters are set into a zone-level family node. The parameters are applied to all zones matching the family.

The `.Solver#motion` uses the elsA attributes for the motion, it applies these attributes to all zones that match the family.



```
MRB                Family_t            MT
  FamilyBC         FamilyBC_t          C1  UserDefined
  .Solver#Motion   UserDefinedData_t   MT
    interpol       DataArray_t         I4  1
    interpol_tool  DataArray_t         C1  adt
    motion         DataArray_t         I4  1
    axis_pnt_x     DataArray_t         R8  0.0
    axis_pnt_y     DataArray_t         R8  0.0
    axis_pnt_z     DataArray_t         R8  0.0
    axis_vct_x     DataArray_t         R8  0.0
    axis_vct_y     DataArray_t         R8  0.0
    axis_vct_z     DataArray_t         R8  1.0
    omega          DataArray_t         R8  0.0
    transl_speed_x DataArray_t         R8  0.0
    transl_speed_y DataArray_t         R8  0.0
    transl_speed_z DataArray_t         R8  0.0
```

Fig (30) Zone motion family

The `.Solver#param` is used to set any attribute you want. This is a straightforward translation to elsA.

5. INTEROPERABILITY FEATURES

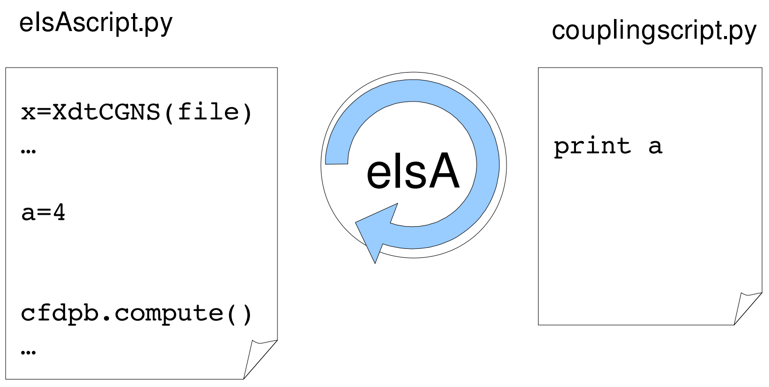
The elsAxdt interface is also during run-time. Some parallel and code-coupling features can be used through elsAxdt, there is no extra features in the module itself but only those implemented in the solver. In other words, you cannot add a feature such as making a BC change its state by reading a CGNS tree unless the BC is design to do so in the elsA kernel.

The elsAxdt provides the user with a Python interface to access code-coupling data in a Python fashion or, in the case of parallel, a way to distribute data on a CGNS tree structure basis, in a more general frame, this is an interoperabilty feature the user can put in his Python applications.

5.1.1 Overview of Python external script

If you want elsA to exchange Python CGNS/trees (so-called SCOPE trees), you have to run elsA with the `-C`, `-X` and `-P` command line arguments. We will see these options later in this section. These options are setting elsA in the code-coupling mode.

The code-coupling mode forces the solver to suspend a computation and to delegate the control to an external Python script. This external script, or the callback script, has to be written by the application developper as well as the main script of the computation. Once the script is performed, the solver resume its work. You have to set when the solver has to suspend and what kind of run-time data it should read/write.



The elsAscript is executed then at each iteration the coupling script is executed. The two scripts are run by the same elsA, if you define a varaible a in the main script, the variable would be available in the couplingscript. If you change its value, this value is kept from one iteration to the next one.

5.1.2 Example

The code-coupling command-line gives the connection parameters. These are the states where the solver is expected to suspend itself, the name of the connection and the script to run while the solver waits. The *name* of your code-coupling connection would be used by the solver for its internal use. For example, this is a command line argument for a code-coupling computation with a legacy script:

```
elsA -X 10 -C xdt-runtime-tree -P A06ax.py -- t.py
```

The same computation with a full CGNS computation would be run with the following command line (Python is now the interpretor instead of elsA):

```
python A06a.py -X 10 -C xdt-runtime-tree -P A06ax.py
```

And the *A06a.py* script itself should contain:

```
import elsAxdt as E
import sys

# ... mode code here
E.args(sys.argv)
```

```

# ... mode code here
mytree=E.XdtCGNS("FluidPlate.cgns").load(E.READ_ALL)
mytree.compute()

# ... mode code here
```

The important line in this script is the *args* method call, with the command line arguments passed to the solver as it starts its computation. These arguments are required by the sopver, but it doesn't care how you give it. For example, you can give your arguments as hard-coded string values:

```
import elsAxdt as E
import sys

E.args(['-X','10','-C','xdt-runtime-tree','-P','A06ax.py'])

mytree=E.XdtCGNS(sys.argv[1]).load(E.READ_ALL)
mytree.compute()
```

Here, the filename is passed as arg while the other values are built in a simulated command line arguments list.

5.2 Code-coupling user interface

5.2.1 Command line arguments

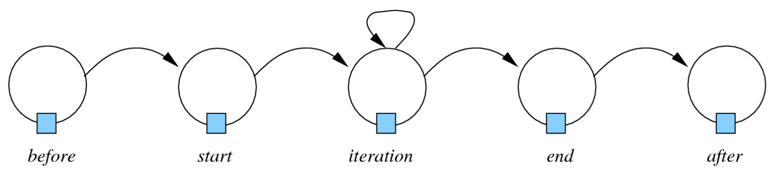
The solver has code-coupling dedicated command line options. These options have to be specified if you want to use the solver for a connection and a data exchange with another software.

-P <filename>.py	Gives the name of the Python file to import at each state stop.
-C <connection-name>	Sets the name for the code-coupling connection. One should use this string to identify the data in the coupling script. In the case each solver instance produces the same sub-tree, the connection name is used as key for identification. The python function ccname returns its value.
-X <state-stop-flag>	A bit-string flag indicating the state stops. This is an hexadecimal OR between values (see the code-coupling examples).
-D	Sets the debug mode *ON*.

5.2.2 State stops

The code-coupling interface is specified by the input and output data structure and the time at which this exchange takes place. The finite state machine (FSM) described in next figure shows the very simple automata of the solver interface. The states are:

☐ *before*	before any processing in the solver
☐ *start*	before the first iteration
☐ *iteration*	before any iteration
☐ *end*	after the last iteration
☐ *after*	after all processings in the solver



Each time the solver passes from one state to another, it checks if there is a request to stop before entering the new state. If there is no stop, the computation continues. If there is a requested stop, the Python script specified as argument of the -P command line argument is run. The script is run in the same name space as the solver and the main script argument. Thus, you can share variables in these two scripts, the three are the same process.

Some other states are specific to dedicated coupling computations, such as the multi-stages turbomachinery computations. Two more states are used:

- ☐ *pcmwrite* after the solver writes data in the xdt-runtime-tree.
- ☐ *pcmread* after the solver reads data in the xdt-runtime-tree. In the case of the multi-stage computation, the user does not have to request a stop at these states. The solver forces a stop at the *pcmwrite* state when it detects the use of a multi-stage boundary condition.

The states actually are set using a bit-string, each state is a bit in this string and the complete string gives the stop behavior of the solver:

before	*0x0001*
after	*0x0002*
start	*0x0004*
end	*0x0008*
iteration	*0x0010*
pcmwrite	*0x0040*
pcmread	*0x0080*

5.3 Coupling-script

The code-coupling script is the piece of Python code executed when the solver reaches a coupling state. A coupling state is declared by the user as a state where the solver has to stop (see command line arguments in the previous section).

The coupling-script can either be a separate script or a function. Such a function, so-called Callback is declared and set into a special Python dictionnary. The code snippet below shows such a declaration:

```
import elsAxdt  as X

a="MY COMPUTATION [%s]"

def myIterationCallBack():
  if (X.iteration() > 1):
   xw=X.xdt(E.PYTHON,("Foo",None))
   xp=xw.dump(E.PYTHON,E.OUTPUT_TREE,"")
   print xp

X.StateCallBack['iteration']=myIterationCallBack

c=X.XdtCGNS("Input.cgns")
c.mode=X.READ_ALL
r=c.compute()
c.save("Output.cgns")
```

The function myIterationCallBack will be run at every iteration state. In that case of a function callback, the user does not have to set the -X command line option, because elsAxdt knows it has to stop at iteration state as soon as it finds a function in its StateCallBack dictionnary.

5.4 Memory ownership

The goal of Xdt is to give access to the internal data of the solver. The solver memory zones (i.e. arrays) are read/write by means of pointers and there is no memory duplication whenever possible. In other words, the array you have in an Xdt tree node is the actual array used by the solver. It is important to understand how the ownership of this memory zone is managed, in particular when you use Python input/output. The Python arrays are numpy arrays. We give some scenario hereafter, please read carefully these scenario to understand how to actually free the Xdt memory.

5.4.1 From Xdt to Python

First scenario, you have a solver array you want to retrieve in Python. You want Xdt to dump the C++ data structure in Python, but you do not want to duplicate memory. During this dump operation, elsAxdt sets the *NPY_OWNDATA* to zero for the large data arrays. These large data arrays are of DataArray_t type in CGNS. Once the *NPY_OWNDATA* is unactivated, Python knowns it cannot release the array memory. It can release the objects, the lists, strings, all other objects created during the dump but not the memory zone of the array. Because it is shared by the solver. Any attempt to a del call to the Python tree you get by the dump (e.g. *get('xdt-output-tree')*) will release Python objects, not *DataArray_t* memory zones and not Xdt objects.

5.4.2 From Python to Xdt

Second scenario, you have a Python array you allocate by yourself, using a Python function that feeds a numpy array. The memory zone is allocated by numpy, if you call del on this array it eventually deallocates the memory (see Reference Counting below). If you pass this array to the fluid solver, you perform a load in elsAxdt (e.g. *XdtPython('xdt-runtime-tree',pytree)*) and a complete Xdt tree is created. This tree shares the memory zones of the large arrays. In that case, not only the *DataArray_t* are shared but all the numpy arrays. This point means you cannot update an existing Xdt tree, you only create new trees.

When Xdt loads the Python tree, all the numpy arrays have their *NPY_OWNDATA* flags set to zera (false). Thus, again, the Python object cannot release the memory of its array, even if the array has been allocated by numpy itself.

5.4.3 Actual Xdt memory release

We know now that when you dump or load an Xdt tree to/from Python, the numpy arrays are no more owner of the memory zone. The only way to free this memory is to release the Xdt tree using the free method. This call deletes all the Xdt tree from the root node, including all the node values and the array memory zones.

We can find some exceptions to this internal free of memory. It is in the case the solver doesn't want Xdt to free the array. For example if you have a direct access to an internal array used for coordinates, a boundary condition or other data that should be protected from unwanted free. These cases so-called Kernel Ownership are quite rare.

5.4.4 Reference Counting

Now, the memory used by the arrays is released (or not), and you cannot see your process memory decrease. We need to detail two points here. First, Python manages reference counting on objects. When you perform a del call, this decrements the counter and when the counter reaches zero, Python marks the object as no more used. The reference counter is incremented each time you refer to the object, for example when you add the same object in two lists, or when you use a new variable to point to an existing object.

An object which is no more used is freed, unless you have defined a specific behavior for the object.

5.4.5 Code-coupling usual scenario

During code-coupling computation, the user will read data from the solver and write data for the solver use. Then, it is mandatory to know how the memory is managed in these code-coupling cases. Unfortunatly, there is no straightforward rule and one should change his policy depending on the data used during code-coupling. Most of the time, small chunks of data are copies and the code-coupling script has to release the data it reads from the solver.

This is true for boundary based code-coupling for example.

The elsAxdt module cannot manage automatically memory release, because only your application knowns when arrays are no more useful. We have the free method to free the Xdt data including the actual memory zones of the arrays, you have the del Python method to decrement Python reference counters and eventually free the objects. It is now your responsability to manage the memory in your Python code-coupling script.

5.4.6 Memory release examples

In the first example below, we show a coupling-script with an Xdt tree that is only used for display. Then it is released. In that case, the tree is an output tree which structure has been defined by the kind of outputs requested by the user. The del call is not really useful, because the tree variable has a global scope and it will be overwritten at the next call to the coupling-script. Then, it would be up to Python to make the tree variable point to the newly created Python list coming from the get call. The previous Python list created in the rpevious coupling-script call will see its reference counter decrease and the list will be freed if no other Python object is using it.

```
import elsAxdt  as X
tree=X.get('xdt-output-tree')
print tree
X.free('xdt-output-tree')
del tree
```

Now you understand that if you call free before the print, the result will be a memory error when you reach the numpy arrays that have no more allocated memory.

The second example shows a classical MPI send/receive. The send data can be freed, but we really want to keep the same tree structure we had as output.

5.4.7 Overview of parallel

The elsA solver can be run on several processors using the parallel features brought by MPI. A parallel computation is a distribution of the zones on the processors. It is up to the user application to indicate on which processor a zone has to be taken into account.

The input tree for a given processor should contain at least the zones taken into account by this processor. The other zones are not required. However, an easy way to pass a correct tree to all processors is to give them the whole CGNS tree and a list of selected zones. The elsAxdt processing would only create the zones related to the target processor.

The output tree is restricted to the local zones, that is the zones computed on the current processor. Thus, each processor will produce a sub tree with the local zones, it is up to the user application to gather the trees in a single one with all zones.

5.4.8 Typical script start-up

The usual pattern in a parallel code using MPI is to identify the current processor and select the zones to compute on this processor:

```
import elsAxdt as e
(rank,size)=e.ranksize()

if (rank == 0):
  zlist=["Zone-001","Zone-002","Zone-003","Zone-004"]
else:
  zlist=["Zone-005","Zone-006","Zone-007","Zone-008"]
```

In this example we know we have 8 zones and we know their names. One would do something more general by reading the CGSN tree, find the zones and their sizes, balance the zones with respect to the processor number and then distribute the zone lists on these processors.

The actual distribution can be done in migration mode or not. In the case of the migration, a zone list is given to elsAxdt but the process number for each zone is given after the CGNS tree has been read:

```
c=e.xdt(e.CGNS,"../Data/SquaredNozzle-06-R.%s"%ext)
c.mode=c.mode | e.READ_OUTPUT
c.action=e.TRANSLATE
c.selection=zlist
c.compute()
c.blocknode("Zone-001",0)
c.blocknode("Zone-002",0)
c.blocknode("Zone-003",0)
c.blocknode("Zone-004",0)
c.blocknode("Zone-005",1)
c.blocknode("Zone-006",1)
c.blocknode("Zone-007",1)
c.blocknode("Zone-008",1)
```

You read here the zlist list of zones per processor is used by the elsAxdt, the selection attribute is set but not the distribution attribute.

5.5 Remarks about data exchange

5.5.1 Shared arrays

The CGNS trees used by elsAxdt are Python trees. These are lists of lists, with Python objects such as string, arrays, lists and these have the same life cycle as any other Python object. The exception are some CGNS tree values used for code-couping or run-time exchange, we detail this exception later in this section

A Python object exists as far as it has a reference on itself. This means that something else is using this object. A variable name is a reference to an object. Thus if you have a variable pointing to a CGNS tree, this one would not be destroyed by Python. If you want Python to destroy the tree and get back the memory used by this tree, you have to explicitely delete the variable, i.e. use the del Python function to destroy the variable.

The Python/CGNS node values are arrays. These arrays are pointing to actual memory allocated by elsAxdt or elsA depending on this is an input or an output CGNS tree. In the case you delete the tree, the arrays are deleted because the memory ownership is set to the CGNS tree, not elsAxdt ot elsA. There is one exception, that is for run-time CGNS tree.

5.5.2 Reserved CGNS tree names

You do not have to set the tree names if you use usual elsAxdt script patterns. Actually, the input and the output trees would be enough and you would not need to define new names.

The default (and recommanded) tree names are:

- ☐ *`xdt-input-tree`* contains all computation data, such as meshes, connectivity, initialisation solutions, boundary conditions... The tree is read at start time.
- ☐ *`xdt-output-tree`* contains all requested results at end of computation.
- ☐ *`xdt-runtime-tree`* is the exchange tree used for code-coupling, it can be read and write but it is up to the application to make sure the modifications are relevant to the computation.

5.5.3 CGNS/HDF or CGNS/ADF files

The low level storage used for CGNS files is not managed by elsAxdt but by the CGNS library used at link time. In other words, if you link elsAxdt with an ADF-based mid-level library, you will obtain ADF low level storage files. Then, if you want to use HDF5 instead of ADF for your files, please refer to the CGNS library production setup.

When a CGNS file is open, with read or write mode, the file is actually open or created on disk. The usual system constraints have to be remind, for example the file ownership, file access, file quotas, etc...

No control is done by the Xdt.

The CGNS file can contain symbolic links refering other CGNS files. The access control should be done at this level too, furthermore the link refers to a name that could be an absolute or a relative path, leading to common problems related to opening files in a current directory.

We suggest to use three levels of files, linked together:

The mesh can be shared by several computations

The computation defines equations, reference state, numerical controls. This file actually has links to the mesh file.

The results which refers to the previous file.

You can also add a level for the initialisation fields. Such a linked-files design leads first to a better access time during the read, but it also insures a concurrent read of meshes and an exclusive write of the solutions at the same time.

5.5.4 SCOPE

The SCOPE initiative is an *Open Specification* of a Python implementation of a CGNS tree.

A SCOPE tree is a Python tree composed of lists of lists. A node is a list and each node has a list of children. This children list is composed of nodes, which are lists... and so on. A node has four objects, a string, a value, a children list and a type.

- Name : A Python string, should not contain `'/'` or a single `'.'` and should not be empty. Except these requirements, a name is whatever you want. Some names are reserved by the CGNS standard (e.g. *`CGNSLibraryVersion`*, *`GridCoordinates`*...)
- Value: A value is a numpy array of type I4,I8,R4,R8 or C1.
- Children: a list of nodes
- Type: A Python string containing the CGNS label (the CGNS type).

The SCOPE tree defines the root node of a tree as `[None,None,[ <children> ], 'CGNSTree_t']` which is not standard.

All elsAxdt trees are SCOPE trees.

6. PACKAGE

6.1 Production and installation

The elsAxdt parser is a true Python module. It is build and installed with the Python standard Distutils. The package contains pure python scripts, C code source files, CGNS data files, various scripts, tests...The actual installation is performed by Python itself, thus it is performed with the requirements you have set for your Python installation.

6.1.1 The package structure

The Python main tree structure has three directories: Doc contains this documentation source files, pdf and html versions of the current release. Misc directory has useless file for end-user. Then elsAxdt is the actual module. The first level of the directory and the utils have the pure Python files. The demo directory is not installed but should be useful for a first try with elsAxdt. The src contains the C and C++ source files for the elsAxdt drivers.

The test directory is not installed, it provides a Go script to run the tests. Read the README file for more information on passing tests on your own platform.

6.1.2 Production requirements

The elsa solver is required. You have to build and install elsa. The production type and the location where you have installed elsa are passed to elsAxdt production by means of the usual elsa shell variable:*ELSAPROD*, *ELSADIST*.

As the elsa libraries are used, the consistency between your elsa, Python and then elsAxdt installations should be insured. In particular, the elsAxdt module requires the double precision for float numbers (r8) andthe shared libraries (so). The use of MPI is also strongly recommanded while not mandatory.

When elsa is built and installed, it generates and copies a configuration file named elsAconfig.py into the directory *$ELSADIST/Dist/bin/$ELSAPROD* (if you have a specific elsa build process, please contact elsa support).The configuration file is a Python script used by elsAxdt to set its own parameters. In case of production problem on your platform, please first look at this *elsAconfig.py* file. We give some details about its contents with the example below, but please do not change values in this file unless you know what you are doing.

```
class eCONFIG:
  def __init__(self):
    self.majorversion = 3
    self.minorversion = 3
    self.production   = '04g'
    self.platform     = "intelx86_mpi_so"
    #
    self.mpi          = True
    self.shared       = True
    self.pynum        = "numpy"
    self.libs         = ['eDescp','eFact','eObf','eTmo','eRhs','eLhs','eBnd','eJoin','eTur','eSio',
                         'eSou','eFxc','eFxd','eDtw', 'eBlk','eGeo','eEos','eMask','eGlob','eOper',
                         'eTbx', 'eFld','eAgt','ePcm','eDef','eAel','eOpt','eLur','eSplit','eChim',
                         'eXdt']

config=eCONFIG()
config.ccompiler = "g++ -g -fPIC"
config.cflags    = "-D_E_FORTRAN_LOOPS_ -DE_SCALAR_COMPUTER -D_ELSA_COMPILER_INTEL_IA64_
                    -D_E_USE_STANDARD_IOSTREAM_ -DE_RTTI -I/home/tools/include/mpich
                   -U_E_MPI_ -DE_MPI -DMPICH_IGNORE_CXX_SEEK -DE_DOUBLEREAL"
config.cshared   = "icc -shared"
config.shflags   = "-fPIC -zmuldefs"
config.extralibs = ['m','rt','dl','pthread','util','mpich','ifcore','stdc++','imf','irc']
config.extralpath= ['/home/tools/local/x86_64/lib','/opt/intel/fce/9.0/lib']
```

6.1.3 Procedure

You have to run the following command to produce the module:

```
export ELSAPROD
export ELSADIST

ELSAPROD=IA64_mpi_so
ELSADIST=/usr/local/tools

python setup.py build
```

The installation can be performed in a user-defined directory. However, we suggest to avoid the installation in the Python installation directories but rather to use the elsa installation directories. As the packagecontains some machine dependant shared libraries, the best is to install the whole set of files using the command line:

```
python setup.py install --prefix=$ELSADIST/Dist/bin/$ELSAPROD
```

You can produce the package with your own set of search directories and library names. The following snippet shows the command line specific options.

```
elsAxdt specific options:
  see README file for details on installation options:
  --[help,prefix,exec-prefix,install-base]
  --xdt-elsA-path=path
  --xdt-elsA-prod=production
  --xdt-python-array=['numpy','numarray','Numeric']
  --xdt-path-includes=path1:path2:path3
  --xdt-path-libs=path1:path2:path3
  --xdt-libs=lib1:lib2:lib3
```

6.2 Installation and run-time problems

6.2.1 Dynamic libraries

The elsAxdt package gathers CGNS, elsa (with Python) and a Python array package. Thus, you have to install all these libraries in your own platform and to define a correct run-time environnement. A common issue is to usedifferent packages version at compile-time and run-time. Please take care of the installation and the run-time environment.

6.2.2 Python libraries

The package has to be built with a Python array module. This can benumpy, numarray, Numeric, but we strongly recommand the use of numpy.

You have to insure that the Python scripts you are using actually import the same array module as the one used during elsAxdt production.

6.2.3 Parallel mode

There are some usual pitfalls, most of them are quite simple to solve once you understand what's happening. We give you there a bit of advice.

First, you may try to run a elsa in parallel mode without using mpirun:

```
>>> import elsAxdt
### elsAxdt - v5.3
### elsAxdt - using numpy
### elsAxdt - standard elsA startup
mpirun must be used to launch all MPI applications
```

Unless you use some specific flavours of MPI, you should use mpirun:

```
mpirun -np 1 elsA.x -c 'import elsAxdt;print elsAxdt.ranksize()'
```

6.2.4 Post-processing common pitfalls

The elsAxdt module does not save the CGNS result by itself. You have to explicitely ask for a disk save. The CGNS result tree is in memory and is lost at end of computation.

The input CGNS tree is not copied into the result tree. For example you won't find the grid coordinates in the result tree unless you explicitely ask elsa to produce such an output.

6.2.5 Current version fingerprint

The elsAxdt version number appears in the standard output at startup. You can also find this version number in the package sources: *`elsAxdt/version.py`* (major and minor version numbers).

The test report has been generated for the elsAxdt module, the elsAxdt/test directory contains a test suite that produces atest report. This report is stored as a file and is released with thedistribution tarball. You re-generate the test report, by running yourself the test suite on your platform.

Some of the test are taken from our customers' usual application (without proprietary data of course). You can refer to the remarks in section find out the specific application pattern you are looking for.

6.3 Known issues and common troubleshootings

The list below gives the issues or limitation known at release time.

6.3.1.a- CGNS tree not taken into account

As the *elsAxdt* parser gain features, it can read new CGNS structures. If you want to known which structure you actual parser is able to understand, please refer to the footnotes in the section The *elsA* Profile and check your parser version.

6.3.1.b- File read fails

Check your file name is correct, in the case of CGNS links, check all the files are in the directory where you actually run the computation. You can also check the `ADF_LINK_PATH`. This variable contains the list of directories where CGNS is going to look for files, the syntax is the same as the usual `PATH`, `PYTHONPATH` or `LD_LIBRARY_PATH`.

6.3.1.c- No way to import modules

Sometimes we have a weird behavior when the `PYTHONPATH` ends with a trailing colon character `':'` (for example `PYTHONPATH=/home/tools/local:/usr/local:`). You remove the trailing colon character and try again.

6.3.1.d- No coupling script check

In coupling mode, the coupling script file name is not checked. If you give a bad file name, you have a weird error such as:

```
  File "CoupingStop.py", line 1
    `?
     ^
SyntaxError: invalid syntax
```

6.3.1.e- Core dump while Python array access

Check the Python array package you use in your Python scripts, it should be the same as the one used by *elsAxdt* (see *elsAxdt* startup message).

6.3.1.f- Grid coordinates are missing in output file

If you ask for grid coordinates in the result CGNS tree, the coordinates are stored in the FlowSolution sub-tree, not the GridCoordinates.

6.3.1.g- The solver crashes when he leaves the last line of Python script

If you use SWIG generation for elsA, check your version and in particular the options `-nodefaultctor -nodefaultdtor` (that should be set).

6.3.1.h- Installation does not match build

You should call the build and the install installation commands with the same command line options. Otherwise, the install would take the default values of the setup, which were overriden by your command line options during the build.

6.3.1.i- Avoid attribute errors using name scopes

You can improve the readability and the portability of your Python code using the `import XXX as X` statement, instead of the simple `import *`. However, be sure you cannot have name collision.

```
# never do this
from elsAxdt import *
```

Using a single letter as new module name can cause troubles, which could have unexpected effect on large or imported scripts as described in the **bad** script below.

```
# this really leads to big problems...
import elsAxdt as r
r.xdtCGNS("010Disk.cgns").load()
(r,s)=r.ranksize() # oups! Overwrite previous Python object named r with an integer !
```

The use of so-called scope-prefix leads to a better readability and reduces name scope problems.

7. REFERENCE TABLES

7.1 Warnings and errors

The solver actually reads the CGNS tree without performing any control. The user has to apply checking and CGNS compliance tools before the solver submission. Thus, most solver errors related to CGNS (or XDT/CGNS) are forced as FATAL errors. The table hereafter describes the possible troubleshootings you may have.

7100	The XDT/CGNS tree you gave as argument is empty. This error can be raised if (1) there is no CGNSBase_t node as father of all other nodes or (2) you did not ask to read the mesh.
7101	No Zone_t node has been found in the CGNSBase_t node of your tree. Check your tree, the solver reads only the first CGNSBase_t and you may have a first empty base before the actual base.
7120	One of the boundaries you gave is not known. The solver cannot find a translation of the given BCType_t into one of its own boundary condition types. Check the BCTypes, the error message gives you the path to the bad BC.
7121	While reading a BC, the required fields where not found. Or, while reading a BC, one of the required field found was incorrect.

7.2 Variable name mapping

The mapping tables are printed when you use the trace option while reading any CGNS file. The list here after details the most used variables.

CGNS name or CGNS-like name	elsA name	Notes
CoordinateX	x	
CoordinateY	y	
CoordinateZ	z	
Density	ro	
MomentumX	rou	
MomentumY	rov	
MomentumZ	row	
MomentumX	rovx	
MomentumY	rovy	
MomentumZ	rovz	
EnergyStagnationDensity	roe	
TurbulentSANuTilde	k	1
TurbulentSANuTildeDensity	rok	1,3
TurbulentEnergyKinetic	k	2
TurbulentDissipation	eps	2
TurbulentDissipationRate	eps	2,3
TurbulentEnergyKineticDensity	rok	2,3
TurbulentDissipationDensity	roeps	2,3
RSDDensityRMS	residual_ro	3
RSDMomentumXRMS	residual_rou	3
RSDMomentumYRMS	residual_rov	3
RSDMomentumZRMS	residual_row	3
RSDEnergyStagnationDensityRMS	residual_roe	3
VelocityX	u	
VelocityY	v	
VelocityZ	w	
ViscosityMolecular	visclam	
ViscosityEddy	viscturb	
Viscosity_EddyMolecularRatio	viscrapp	
SkinFrictionX	frictionvectorx	
SkinFrictionY	frictionvectory	
SkinFrictionZ	frictionvectorz	
SkinFrictionMagnitude	frictionmodulus	
TurbulentDistance	walldistance	
TurbulentDistanceIndex	wallglobalindex	3
Mach	m	
Pressure	p	
PressureStagnation	stag_pressure	
EnthalpyStagnation	stag_enthalpy	
NormalHeatFlux	normalheatflux	
Temperature	tsta	
TemperatureStagnation	tgen	

CGNS name or CGNS-like name	elsA name	Notes
MomentumXFlux	flux_rou	3
MomentumYFlux	flux_rov	3
MomentumZFlux	flux_row	3
TorqueX	torque_rou	
TorqueY	torque_rov	
TorqueZ	torque_row	
WallCellSize	yplusmeshsize	
BladeSectionForce	bladesecforce	3
BladeSectionForceX	bladesecforcex	3
BladeSectionForceY	bladesecforcey	3
BladeSectionForceZ	bladesecforcez	3
BladeSectionTorque	bladesectorque	3
BladeSectionTorqueX	bladesectorquex	3
BladeSectionTorqueY	bladesectorquey	3
BladeSectionTorqueZ	bladesectorquez	3
BladeSectionForceSquareMach	bladesecforcem2	3
BladeSectionForceXSquareMach	bladesecforcem2x	3
BladeSectionForceYSquareMach	bladesecforcem2y	3
BladeSectionForceZSquareMach	bladesecforcem2z	3
BladeSectionTorqueSquareMach	bladesectorquem2	3
BladeSectionTorqueXSquareMach	bladesectorquem2x	3
BladeSectionTorqueYSquareMach	bladesectorquem2y	3
BladeSectionTorqueZSquareMach	bladesectorquem2z	3
IterationNumber	iteration	3
RadialPosition	ronr	3
ChordLength	chord	3
BladeAzimuth	azimuth	3
ChimeraCellType	ichim	3
VortexGeneratorForce	vortexgeneratorforce	3
VortexGeneratorLocation	vortexgeneratorlocation	3

Notes:

1. Available only for Spalart turbulence model, the elsA variable name doesn't reflect the actual value used for output.
2. Available for K-* turbulence models.
3. Not a CGNS name as described in Annex A of CGNS/SIDS.

7.3 Tutorial contents

The Tutorial files are small test cases which only illustrate the elsAxdt use. The tutorial has cgns files you can use as patterns and script files. Please note the striked through file names are available but cannot be used with the current elsAxdt version. These will be useful for the next release.

Some test cases look the same or are using the same features. We do not remove these similar tests, you just have to know that, in some cases, you may have more than one way to do the same thing.

A test case has its input `file` in `Tutorial/Data/XXXDisk.cgns` (with `XXX` your test case number). The Python script to run is `Tutorial/Script/XXX<script title>.py` and the output is set in `Tutorial/Output/XXX<Script title>.cgns` or similar.

The `Tutorial/Reference` contains the expected output for all cases, you can pass all tutorial cases as elsAxdt test suite and compare with reference. The `Tutorial/Script/coverage.py` script performs such a coverage.

Tutorial files

001	Simple computation.
002	A **simple output definition** reused as linked-to internal `FlowSolution#EndOfRun` nodes.
003	An **family based output** definition made using the `Family_t` nodes .
004	A **surface output** definition distributed on `BC_t` nodes.
005	An output defined with `FlowSolution#EndOfRun` and producing a **writing** solution file for **restart**
006	A file with a CGNS symbolic link to `../Output/005WriteInitSolution.cgns` as `FlowSolution#Init` to **read** for a **restart**. Couldn't be used without this linked-to file.
007	Two **different output** for an integration based on a **same family**.
008	Simple **parallel** computation with 2 MPI processes.
009	Simple **chimera** case.
010	Simple **chimera writing** interpolation and masking data as **non-CGNS** separate files.
011	Simple **chimera reading** interpolation and masking data as **non-CGNS** separate files.
012	Simple **chimera reading** interpolation and masking data as **CGNS** separate files.
013	A chimera computation with a **doubly defined** boundary condition.
~~014~~	**Multiple reference states** used for both initialisation and BCs.
015	**User defined Grid Location** for `FlowSolution_t`.
~~016~~	Ouput to get **GridCoordinates** in absolute frame as `FlowSolution_t` and `GridCoordinates_t`.
~~017~~	So-called **coarsen mesh** example, grid coordinates are read every other point.
~~018~~	Output to get an unsteady or **time dependant** as final result of the computation.
~~019~~	Case of a skeletton file **re-using links** ar restart.
020	**NCT case**: with CellListDonor as PointRange, mono base.
021	**NCT case**: without CellListDonor, mono base.
022	**NCT case**: without CellListDonor, multi base.
023	**NCT case**: with PointList, multi base.
~~024~~	**Rigid motion** on the zone-level family
~~025~~	**Block parameters** on the zone-level family
~~026~~	Case with a **Vortex generator write**
~~027~~	Case with a **Vortex generator read** using the case 026.
~~028~~	Use of the **GoverninEquationSet** tree
~~029~~	A **2D CGNSBase** test case
030	Complex case with an **external update** using Python for a **NREF** boundary limit.
031	**Protocol** based output for BC values, related to **code-coupling** use.
032	Output with **bl_quantities** (boundary layer).
033	Writes only the **conservative** values for **restart**.
034	Reads only the **conservative** values for **restart**.
035	A so-called **cellfict** output.
036	Output of **ichim** on **BC** defined surface.
037	Writes only the **conservative** and the **wall distance** values for **restart**.
038	Reads only the **conservative** and the **wall distance** values for **restart**.
039	Writes only the **wall distance** values for **restart**.
040	Reads only the **wall distance** values for **restart**.
~~041~~	**Parallel** computation with **per-process** file **write**.
~~042~~	**Parallel** computation with **per-process** file **read**, each file is **incomplete** so that each process has not all the CGNS

	tree available.
~~043~~	**Parallel** computation with a **multi-base** family **integration**.
~~044~~	**Migration** case reading **only** the **mesh**.
~~045~~	**Migration** case with CGNS **attributes overwrite**.
~~046~~	**Migration** case with a **Python** definition of computation **attributes**.
~~047~~	**Migration** case with a **Python** definition of **CFL function**.
~~048~~	**Migration** case with **aeroelastic** computation.
~~049~~	Case with **grid** arrays **shared** in memory.
~~050~~	Case with **unsteady** data arrays **shared** in memory.
~~051~~	**Asynchroneous** data exchange case.
~~052~~	**On the fly** storage of convergence data.
~~053~~	External use of **pyMPI** module.
~~054~~	**BC input control** with external script.
~~055~~	**BC actuator** disk .
~~056~~	**BC periodic.**
~~057~~	**BC input** with user defined data on **BC-level family.**
~~058~~	**BC mobile_coef**
~~059~~	**BC Froude**
~~060~~	Different ways to define **grid location** output using **user defined** data.
~~061~~	Different ways to **write grid location** output using **GridLocation**.
~~062~~	Different ways to **read grid location** output using **GridLocation**.
~~063~~	**Read CGNS transition** information.
~~064~~	**Write CGNS transition** information.
~~065~~	**Write CGNS Chimera** information.
~~070~~	Adjoint definitions as input.
~~071~~	Adjoint definition for output.
~~072~~	Unstructured mesh and connectivity read.
~~073~~	Unstructured solution write.

CONTENTS

FIGURES